starting out with >>>

# JAVA™

## From Control Structures through Data Structures

4TH EDITION

# STARTING OUT WITH

# JAVA™

## From Control Structures through Data Structures

This page intentionally left blank

# STARTING OUT WITH
# JAVA™

## From Control Structures through Data Structures

FORTH EDITION

**Tony Gaddis**
Haywood Community College

**Godfrey Muganda**
North Central College

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

**Senior Vice President Courseware Portfolio Management:** Marcia J. Horton
**Director, Portfolio Management: Engineering, Computer Science & Global Editions:** Julian Partridge
**Portfolio Manager:** Matt Goldstein
**Portfolio Management Assistant:** Meghan Jacoby
**Product Marketing Manager:** Yvonne Vannatta Field
**Marketing Manager:** Demetrius Hall Marketing
**Assistant:** Jon Bryant
**Managing Content Producer:** Scott Disanno Content
**Producer:** Amanda Brands
**Inventory Manager:** Bruce Boundy Manufacturing
Buyer, Higher Ed, Lake Side Communications,
**Inc. (LSC):** Maura Zaldivar- Garcia Cover
**Designer:** Joyce Wells
**Cover Image:** phloen/Alamy Stock Photo
**Manager, Rights Management:** Ben Ferrini
**Full-Service Project Management:** Sasibalan Chidambaram, SPi Global Composition: SPi
**Global Cover Printer:** Phoenix Color/Hagerstown
**Printer/Bindery:** LSC Communications, Inc.

Copyright © 2019, 2016, 2012, 2007 by Pearson Education, Inc. or its affiliates. All Rights Reserved. Printed in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit www.pearsoned.com/permissions/.

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Unless otherwise indicated herein, any third-party trademarks that may appear in this work are the property of their respective owners and any references to third-party trademarks, logos or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc. or its affiliates, authors, licensees or distributors.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability. Whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract. Negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time partial screen shots may be viewed in full within the software version specified.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.
Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

**Library of Congress Cataloging-in-Publication Data**
Gaddis, Tony, author.
Starting out with Java. From control structures through data structures /Tony Gaddis,
Haywood Community College, Godfrey Muganda, North Central College.
Description: Fourth edition. | Pearson, [2019] | Includes bibliographical references and index.

Identifiers: LCCN 2018002219| ISBN 9780134787961 | ISBN 013478796X
Subjects: LCSH: Java (Computer program language) | Data structures
(Computer science)
Classification: LCC QA76.73.J38 G33 2019 | DDC 005.13/3--dc23 LC record
available at https://
lccn.loc.gov/2018002219

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN-13: 978-0-13-478796-1
ISBN-10: 0-13-478796-X

# Contents in Brief

Chapter 1 **Introduction to Computers and Java** 1
Chapter 2 **Java Fundamentals** 27
Chapter 3 **Decision Structures** 111
Chapter 4 **Loops and Files** 189
Chapter 5 **Methods** 269
Chapter 6 **A First Look at Classes** 317
Chapter 7 **Arrays and the `ArrayList` Class** 403
Chapter 8 **A Second Look at Classes and Objects** 493
Chapter 9 **Text Processing and More about Wrapper Classes** 557
Chapter 10 **Inheritance** 611
Chapter 11 **Exceptions and Advanced File I/O** 701
Chapter 12 **JavaFX: GUI Programming and Basic Controls** 759
Chapter 13 **JavaFX: Advanced Controls** 823
Chapter 14 **JavaFX: Graphics, Effects, and Media** 909
Chapter 15 **Recursion** 999
Chapter 16 **Sorting, Searching, and Algorithm Analysis** 1027
Chapter 17 **Generics** 1079
Chapter 18 **Collections and the Stream API** 1125
Chapter 19 **Linked Lists** 1195
Chapter 20 **Stacks and Queues** 1245
Chapter 21 **Binary Trees, AVL Trees, and Priority Queues** 1287
**Index** 1353

**Appendices A–M** Companion Website
**Chapters 22–25** Companion Website
**Case Studies 1–7** Companion Website

# Contents

**Preface xxv**

Chapter 1 **Introduction to Computers and Java 1**

1.1 Introduction . . . . 1
1.2 Why Program? . . . . 1
1.3 Computer Systems: Hardware and Software . . . . 2
*Hardware* . . . . *2*
*Software* . . . . *5*
1.4 Programming Languages . . . . 6
*What Is a Program?* . . . . *6*
*A History of Java* . . . . *8*
1.5 What Is a Program Made Of? . . . . 8
*Language Elements* . . . . *8*
*Lines and Statements* . . . . *11*
*Variables* . . . . *11*
*The Compiler and the Java Virtual Machine* . . . . *12*
*Java Software Editions* . . . . *13*
*Compiling and Running a Java Program* . . . . *14*
1.6 The Programming Process . . . . 16
*Software Engineering* . . . . *18*
1.7 Object-Oriented Programming . . . . 19
*Review Questions and Exercises* 21
*Programming Challenge* 25

Chapter 2 **Java Fundamentals 27**

2.1 The Parts of a Java Program . . . . 27
2.2 The `print` and `println` Methods, and the Java API . . . . 33
2.3 Variables and Literals . . . . 39
*Displaying Multiple Items with the + Operator* . . . . *40*
*Be Careful with Quotation Marks* . . . . *41*
*More about Literals* . . . . *42*
*Identifiers* . . . . *42*
*Class Names* . . . . *44*

2.4 Primitive Data Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .44
*The Integer Data Types* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *46*
*Floating-Point Data Types* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *47*
*The `boolean` Data Type* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *50*
*The `char` Data Type* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *50*
*Variable Assignment and Initialization* . . . . . . . . . . . . . . . . . . . . . . . . . *52*
*Variables Hold Only One Value at a Time* . . . . . . . . . . . . . . . . . . . . . . . *53*
2.5 Arithmetic Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .54
*Integer Division* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *57*
*Operator Precedence* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *57*
*Grouping with Parentheses* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *59*
*The `Math` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *62*
2.6 Combined Assignment Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .63
2.7 Conversion between Primitive Data Types. . . . . . . . . . . . . . . . . . . . . . . . . .65
*Mixed Integer Operations* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *67*
*Other Mixed Mathematical Expressions*. . . . . . . . . . . . . . . . . . . . . . . . . . *68*
2.8 Creating Named Constants with `final` . . . . . . . . . . . . . . . . . . . . . . . . . . .69
2.9 The `String` Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .70
*Objects Are Created from Classes*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *71*
*The `String` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *71*
*Primitive Type Variables and Class Type Variables* . . . . . . . . . . . . . . . . . . *71*
*Creating a `String` Object*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *72*
2.10 Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .76
2.11 Comments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .78
2.12 Programming Style . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .83
2.13 Reading Keyboard Input. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .85
*Reading a Character* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *89*
*Mixing Calls to `nextLine` with Calls to Other `Scanner` Methods* . . . . . . . *89*
2.14 Dialog Boxes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .93
*Displaying Message Dialogs* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *93*
*Displaying Input Dialogs*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *94*
*An Example Program*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *94*
*Converting String Input to Numbers* . . . . . . . . . . . . . . . . . . . . . . . . . . . . *96*
2.15 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .99
*Review Questions and Exercises* 100
*Programming Challenges* 106

Chapter 3 **Decision Structures 111**

3.1 The `if` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .111
*Using Relational Operators to Form Conditions*. . . . . . . . . . . . . . . . . . . *113*
*Putting It All Together* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *114*
*Programming Style and the `if` Statement* . . . . . . . . . . . . . . . . . . . . . . . *117*
*Be Careful with Semicolons* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *117*
*Having Multiple Conditionally Executed Statements* . . . . . . . . . . . . . . . *118*
*Flags* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *118*
*Comparing Characters*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *119*

3.2 The `if-else` Statement . . . . . 121
3.3 Nested `if` Statements . . . . . 124
3.4 The `if-else-if` Statement . . . . . 128
3.5 Logical Operators . . . . . 134
*The Precedence of Logical Operators* . . . . . 139
*Checking Numeric Ranges with Logical Operators* . . . . . 140
3.6 Comparing String Objects . . . . . 142
*Ignoring Case in String Comparisons* . . . . . 146
3.7 More about Variable Declaration and Scope . . . . . 147
3.8 The Conditional Operator (Optional) . . . . . 149
3.9 The switch Statement . . . . . 150
3.10 Displaying Formatted Output with `System.out.printf` and `String.format` . . . . . 160
*Format Specifier Syntax* . . . . . 163
*Precision* . . . . . 164
*Specifying a Minimum Field Width* . . . . . 164
*Flags* . . . . . 167
*Formatting String Arguments* . . . . . 170
*The `String.format` Method* . . . . . 172
3.11 Common Errors to Avoid . . . . . 174
*Review Questions and Exercises* 175
*Programming Challenges* 181

## Chapter 4 **Loops and Files** 189

4.1 The Increment and Decrement Operators . . . . . 189
*The Difference between Postfix and Prefix Modes* . . . . . 192
4.2 The `while` Loop . . . . . 193
*The `while` Loop Is a Pretest Loop* . . . . . 196
*Infinite Loops* . . . . . 196
*Don't Forget the Braces with a Block of Statements* . . . . . 197
*Programming Style and the `while` Loop* . . . . . 198
4.3 Using the `while` Loop for Input Validation . . . . . 200
4.4 The `do-while` Loop . . . . . 204
4.5 The `for` Loop . . . . . 207
*The `for` Loop Is a Pretest Loop* . . . . . 210
*Avoid Modifying the Control Variable in the Body of the `for` Loop* . . . . . 211
*Other Forms of the Update Expression* . . . . . 211
*Declaring a Variable in the `for` Loop's Initialization Expression* . . . . . 211
*Creating a User Controlled `for` Loop* . . . . . 212
*Using Multiple Statements in the Initialization and Update Expressions* . . . . . 213
4.6 Running Totals and Sentinel Values . . . . . 216
*Using a Sentinel Value* . . . . . 219
4.7 Nested Loops . . . . . 221
4.8 The break and continue Statements (Optional) . . . . . 229
4.9 Deciding Which Loop to Use . . . . . 229

4.10 Introduction to File Input and Output . . . . . 230
*Using the `PrintWriter` Class to Write Data to a File* . . . . . 230
*Appending Data to a File* . . . . . 236
*Specifying the File Location* . . . . . 237
*Reading Data from a File* . . . . . 237
*Reading Lines from a File with the `nextLine` Method* . . . . . 238
*Adding a `throws` Clause to the Method Header* . . . . . 241
*Checking for a File's Existence* . . . . . 245
4.11 Generating Random Numbers with the `Random` Class . . . . . 249
4.12 Common Errors to Avoid . . . . . 255
*Review Questions and Exercises* 256
*Programming Challenges* 262

## Chapter 5 **Methods 269**

5.1 Introduction to Methods . . . . . 269
*`void` Methods and Value-Returning Methods* . . . . . 270
*Defining a `void` Method* . . . . . 271
*Calling a Method* . . . . . 272
*Layered Method Calls* . . . . . 276
*Using Documentation Comments with Methods* . . . . . 277
5.2 Passing Arguments to a Method . . . . . 279
*Argument and Parameter Data Type Compatibility* . . . . . 281
*Parameter Variable Scope* . . . . . 282
*Passing Multiple Arguments* . . . . . 282
*Arguments Are Passed by Value* . . . . . 284
*Passing Object References to a Method* . . . . . 285
*Using the `@param` Tag in Documentation Comments* . . . . . 288
5.3 More about Local Variables . . . . . 290
*Local Variable Lifetime* . . . . . 292
*Initializing Local Variables with Parameter Values* . . . . . 292
5.4 Returning a Value from a Method . . . . . 293
*Defining a Value-Returning Method* . . . . . 293
*Calling a Value-Returning Method* . . . . . 294
*Using the `@return` Tag in Documentation Comments* . . . . . 296
*Returning a `boolean` Value* . . . . . 300
*Returning a Reference to an Object* . . . . . 300
5.5 Problem Solving with Methods . . . . . 302
*Calling Methods That Throw Exceptions* . . . . . 305
5.6 Common Errors to Avoid . . . . . 305
*Review Questions and Exercises* 306
*Programming Challenges* 311

## Chapter 6 **A First Look at Classes 317**

6.1 Objects and Classes . . . . . 317
*Classes: Where Objects Come From* . . . . . 318

*Classes in the Java API* . . . 319
*Primitive Variables vs. Objects* . . . 321
6.2 Writing a Simple Class, Step by Step . . . 324
*Accessor and Mutator Methods* . . . 338
*The Importance of Data Hiding* . . . 338
*Avoiding Stale Data* . . . 339
*Showing Access Specification in UML Diagrams* . . . 339
*Data Type and Parameter Notation in UML Diagrams* . . . 339
*Layout of Class Members* . . . 340
6.3 Instance Fields and Methods . . . 341
6.4 Constructors . . . 346
*Showing Constructors in a UML Diagram* . . . 348
*Uninitialized Local Reference Variables* . . . 348
*The Default Constructor* . . . 348
*Writing Your Own No-Arg Constructor* . . . 349
*The `String` Class Constructor* . . . 350
6.5 Passing Objects as Arguments . . . 358
6.6 Overloading Methods and Constructors . . . 370
*The `BankAccount` Class* . . . 372
*Overloaded Methods Make Classes More Useful* . . . 378
6.7 Scope of Instance Fields . . . 378
*Shadowing* . . . 379
6.8 Packages and `import` Statements . . . 380
*Explicit and Wildcard `import` Statements* . . . 380
*The `java.lang` Package* . . . 381
*Other API Packages* . . . 381
6.9 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities . . . 382
*Finding the Classes* . . . 382
*Identifying a Class's Responsibilities* . . . 385
*This Is Only the Beginning* . . . 388
6.10 Common Errors to Avoid . . . 388
*Review Questions and Exercises* 389
*Programming Challenges* 394

## Chapter 7 Arrays and the `ArrayList` Class 403

7.1 Introduction to Arrays . . . 403
*Accessing Array Elements* . . . 405
*Inputting and Outputting Array Contents* . . . 406
*Java Performs Bounds Checking* . . . 409
*Watch Out for Off-by-One Errors* . . . 410
*Array Initialization* . . . 411
*Alternate Array Declaration Notation* . . . 412
7.2 Processing Array Elements . . . 413
*Array Length* . . . 415
*The Enhanced `for` Loop* . . . 416

*Letting the User Specify an Array's Size* . . . . . . . . . . . . . . . . . . . . . . . . *417*
*Reassigning Array Reference Variables* . . . . . . . . . . . . . . . . . . . . . . . . . *419*
*Copying Arrays* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *420*
7.3 Passing Arrays as Arguments to Methods . . . . . . . . . . . . . . . . . . . . . . .422
7.4 Some Useful Array Algorithms and Operations
Comparing Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .426
*Summing the Values in a Numeric Array* . . . . . . . . . . . . . . . . . . . . . . . *427*
*Getting the Average of the Values in a Numeric Array* . . . . . . . . . . . . . . *428*
*Finding the Highest and Lowest Values in a Numeric Array* . . . . . . . . . . *428*
*The `SalesData` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *429*
*Partially Filled Arrays* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *437*
*Working with Arrays and Files* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *438*
7.5 Returning Arrays from Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .439
7.6 `String` Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .441
*Calling `String` Methods from an Array Element* . . . . . . . . . . . . . . . . . . *443*
7.7 Arrays of Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .444
7.8 The Sequential Search Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . .447
7.9 Two-Dimensional Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .450
*Initializing a Two-Dimensional Array* . . . . . . . . . . . . . . . . . . . . . . . . . . *454*
*The `length` Field in a Two-Dimensional Array* . . . . . . . . . . . . . . . . . . . *455*
*Displaying All the Elements of a Two-Dimensional Array* . . . . . . . . . . . . *457*
*Summing All the Elements of a Two-Dimensional Array* . . . . . . . . . . . . . *457*
*Summing the Rows of a Two-Dimensional Array* . . . . . . . . . . . . . . . . . . *458*
*Summing the Columns of a Two-Dimensional Array* . . . . . . . . . . . . . . . *458*
*Passing Two-Dimensional Arrays to Methods* . . . . . . . . . . . . . . . . . . . . *459*
*Ragged Arrays* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *461*
7.10 Arrays with Three or More Dimensions . . . . . . . . . . . . . . . . . . . . . . . .462
7.11 The Selection Sort and the Binary Search Algorithms . . . . . . . . . . . . .463
*The Selection Sort Algorithm* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *463*
*The Binary Search Algorithm* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *466*
7.12 Command-Line Arguments and Variable-Length Argument Lists . . . .468
*Command-Line Arguments* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *469*
*Variable-Length Argument Lists* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *470*
7.13 The `ArrayList` Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .472
*Creating and Using an `ArrayList` Object* . . . . . . . . . . . . . . . . . . . . . . . *473*
*Using the Enhanced `for` Loop with an `ArrayList`* . . . . . . . . . . . . . . . . *474*
*The `ArrayList` Class's `toString` method* . . . . . . . . . . . . . . . . . . . . . . *475*
*Removing an Item from an `ArrayList`* . . . . . . . . . . . . . . . . . . . . . . . . . *476*
*Inserting an Item* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *477*
*Replacing an Item* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *478*
*Capacity* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *479*
*Storing Your Own Objects in an `ArrayList`* . . . . . . . . . . . . . . . . . . . . . *479*
*Using the Diamond Operator for Type Inference* . . . . . . . . . . . . . . . . . . *480*
7.14 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .481
*Review Questions and Exercises* 481
*Programming Challenges* 486

## Chapter 8 A Second Look at Classes and Objects 493

8.1 Static Class Members . . . . . . . . . . 493
*A Quick Review of Instance Fields and Instance Methods* . . . . . . . . . . *493*
*Static Members* . . . . . . . . . . *494*
*Static Fields* . . . . . . . . . . *494*
*Static Methods* . . . . . . . . . . *497*
8.2 Passing Objects as Arguments to Methods . . . . . . . . . . 500
8.3 Returning Objects from Methods . . . . . . . . . . 503
8.4 The `toString` Method . . . . . . . . . . 505
8.5 Writing an `equals` Method . . . . . . . . . . 509
8.6 Methods That Copy Objects . . . . . . . . . . 512
*Copy Constructors* . . . . . . . . . . *514*
8.7 Aggregation . . . . . . . . . . 515
*Aggregation in UML Diagrams* . . . . . . . . . . *523*
*Security Issues with Aggregate Classes* . . . . . . . . . . *523*
*Avoid Using `null` References* . . . . . . . . . . *525*
8.8 The `this` Reference Variable . . . . . . . . . . 528
*Using `this` to Overcome Shadowing* . . . . . . . . . . *529*
*Using `this` to Call an Overloaded Constructor from Another Constructor* . . . . . . . . . . *530*
8.9 Enumerated Types . . . . . . . . . . 531
*Enumerated Types Are Specialized Classes* . . . . . . . . . . *532*
*Switching On an Enumerated Type* . . . . . . . . . . *538*
8.10 Garbage Collection . . . . . . . . . . 540
*The `finalize` Method* . . . . . . . . . . *542*
8.11 Focus on Object-Oriented Design: Class Collaboration . . . . . . . . . . 542
*Determining Class Collaborations with CRC Cards* . . . . . . . . . . *545*
8.12 Common Errors to Avoid . . . . . . . . . . 546
*Review Questions and Exercises* 547
*Programming Challenges* 551

## Chapter 9 Text Processing and More about Wrapper Classes 557

9.1 Introduction to Wrapper Classes . . . . . . . . . . 557
9.2 Character Testing and Conversion with the `Character` Class . . . . . . . . . . 558
*Character Case Conversion* . . . . . . . . . . *563*
9.3 More `String` Methods . . . . . . . . . . 566
*Searching for Substrings* . . . . . . . . . . *566*
*Extracting Substrings* . . . . . . . . . . *572*
*Methods That Return a Modified `String`* . . . . . . . . . . *576*
*The Static `valueOf` Methods* . . . . . . . . . . *577*
9.4 The `StringBuilder` Class . . . . . . . . . . 579
*The `StringBuilder` Constructors* . . . . . . . . . . *580*
*Other `StringBuilder` Methods* . . . . . . . . . . *581*

*The `toString` Method* . . . . . . . . . . 584
9.5 Tokenizing Strings . . . . . . . . . . 589
9.6 Wrapper Classes for the Numeric Data Types . . . . . . . . . . 594
*The Static `toString` Methods* . . . . . . . . . . 594
*The `toBinaryString`, `toHexString`, and `toOctalString` Methods* . . . 594
*The `MIN_VALUE` and `MAX_VALUE` Constants* . . . . . . . . . . 595
*Autoboxing and Unboxing* . . . . . . . . . . 595
9.7 Focus on Problem Solving: The `TestScoreReader` Class . . . . . . . . . . 597
9.8 Common Errors to Avoid . . . . . . . . . . 601
*Review Questions and Exercises* 601
*Programming Challenges* 605

## Chapter 10 **Inheritance 611**

10.1 What Is Inheritance? . . . . . . . . . . 611
*Generalization and Specialization* . . . . . . . . . . 611
*Inheritance and the "Is a" Relationship* . . . . . . . . . . 612
*Inheritance in UML Diagrams* . . . . . . . . . . 620
*The Superclass's Constructor* . . . . . . . . . . 621
*Inheritance Does Not Work in Reverse* . . . . . . . . . . 623
10.2 Calling the Superclass Constructor . . . . . . . . . . 624
*When the Superclass Has No Default or No-Arg Constructors* . . . . . . . . 630
*Summary of Constructor Issues in Inheritance* . . . . . . . . . . 631
10.3 Overriding Superclass Methods . . . . . . . . . . 632
*Overloading versus Overriding* . . . . . . . . . . 637
*Preventing a Method from Being Overridden* . . . . . . . . . . 640
10.4 Protected Members . . . . . . . . . . 641
*Package Access* . . . . . . . . . . 646
10.5 Chains of Inheritance . . . . . . . . . . 647
*Class Hierarchies* . . . . . . . . . . 653
10.6 The `Object` Class . . . . . . . . . . 653
10.7 Polymorphism . . . . . . . . . . 655
*Polymorphism and Dynamic Binding* . . . . . . . . . . 656
*The "Is-a" Relationship Does Not Work in Reverse* . . . . . . . . . . 658
*The `instanceof` Operator* . . . . . . . . . . 659
10.8 Abstract Classes and Abstract Methods . . . . . . . . . . 660
*Abstract Classes in UML* . . . . . . . . . . 666
10.9 Interfaces . . . . . . . . . . 667
*An Interface is a Contract* . . . . . . . . . . 669
*Fields in Interfaces* . . . . . . . . . . 673
*Implementing Multiple Interfaces* . . . . . . . . . . 673
*Interfaces in UML* . . . . . . . . . . 673
*Default Methods* . . . . . . . . . . 674
*Polymorphism and Interfaces* . . . . . . . . . . 676
10.10 Anonymous Inner Classes . . . . . . . . . . 681
10.11 Functional Interfaces and Lambda Expressions . . . . . . . . . . 684

10.12 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .689
*Review Questions and Exercises* 690
*Programming Challenges* 696

## Chapter 11 **Exceptions and Advanced File I/O 701**

11.1 Handling Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .701
*Exception Classes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 702
*Handling an Exception* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 703
*Retrieving the Default Error Message* . . . . . . . . . . . . . . . . . . . . . . . . . 707
*Polymorphic References to Exceptions* . . . . . . . . . . . . . . . . . . . . . . . . 710
*Using Multiple `catch` Clauses to Handle Multiple Exceptions* . . . . . . . . 710
*The `finally` Clause* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 718
*The Stack Trace* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 720
*Handling Multiple Exceptions with One `catch` Clause* . . . . . . . . . . . . . 721
*When an Exception Is Not Caught* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 723
*Checked and Unchecked Exceptions* . . . . . . . . . . . . . . . . . . . . . . . . . . 724
11.2 Throwing Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .725
*Creating Your Own Exception Classes* . . . . . . . . . . . . . . . . . . . . . . . . . 728
*Using the `@exception` Tag in Documentation Comments* . . . . . . . . . . . 731
11.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .732
*Binary Files* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 732
*Random Access Files* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 739
*Object Serialization* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 744
*Serializing Aggregate Objects* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 748
11.4 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .749
*Review Questions and Exercises* 749
*Programming Challenges* 755

## Chapter 12 **JavaFX: GUI Programming and Basic Controls 759**

12.1 Graphical User Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .759
*Event-Driven GUI Programs* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 761
12.2 Introduction to JavaFX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .762
*Controls* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 762
*Stages and Scenes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 763
*The `Application` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 763
12.3 Creating Scenes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .765
*Creating Controls* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 766
*Creating Layout Containers* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 766
*Creating a `Scene` Object* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 767
*Adding the `Scene` Object to the Stage* . . . . . . . . . . . . . . . . . . . . . . . . 768
*Setting the Size of the Scene* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 770
*Aligning Controls in an `HBox` Layout Container* . . . . . . . . . . . . . . . . . 770
12.4 Displaying Images . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .772

*Loading Images from an Internet Location* . . . . . . . . . . 775
*Setting the Size of an Image* . . . . . . . . . . 775
*Preserving the Image's Aspect Ratio* . . . . . . . . . . 775
*Changing an ImageView's Image* . . . . . . . . . . 776
12.5 More about the `HBox`, `VBox`, and `GridPane` Layout Containers . . . . . . . 776
*The `HBox` Layout Container* . . . . . . . . . . 777
*The `VBox` Layout Container* . . . . . . . . . . 782
*The `GridPane` Layout Container* . . . . . . . . . . 784
*Using Multiple Layout Containers in the Same Screen* . . . . . . . . . . 791
12.6 `Button` Controls and Events . . . . . . . . . . 792
*Handling Events* . . . . . . . . . . 794
*Writing Event Handlers* . . . . . . . . . . 794
*Registering an Event Handler* . . . . . . . . . . 795
12.7 Reading Input with `TextField` Controls . . . . . . . . . . 799
12.8 Using Anonymous Inner Classes and Lambda Expressions to Handle Events . . . . . . . . . . 803
*Using Anonymous Inner Classes to Create Event Handlers* . . . . . . . . . . 803
*Using Lambda Expressions to Create Event Handlers* . . . . . . . . . . 806
12.9 The `BorderPane` Layout Container . . . . . . . . . . 808
12.10 The `ObservableList` Interface . . . . . . . . . . 812
12.11 Common Errors to Avoid . . . . . . . . . . 814
*Review Questions and Exercises* 814
*Programming Challenges* 818

## Chapter 13 **JavaFX: Advanced Controls 823**

13.1 Styling JavaFX Applications with CSS . . . . . . . . . . 823
*Type Selector Names* . . . . . . . . . . 824
*Style Properties* . . . . . . . . . . 825
*Applying a Stylesheet to a JavaFX Application* . . . . . . . . . . 826
*Applying Styles to the Root Node* . . . . . . . . . . 830
*Specifying Multiple Selectors in the Same Style Definition* . . . . . . . . . . 832
*Working with Colors* . . . . . . . . . . 832
*Creating a Custom Style Class Name* . . . . . . . . . . 834
*ID Selectors* . . . . . . . . . . 836
*Inline Style Rules* . . . . . . . . . . 837
13.2 `RadioButton` Controls . . . . . . . . . . 838
*Determining in Code Whether a `RadioButton` Is Selected* . . . . . . . . . . 839
*Selecting a `RadioButton` in Code* . . . . . . . . . . 839
*Responding to `RadioButton` Clicks* . . . . . . . . . . 844
13.3 `CheckBox` Controls . . . . . . . . . . 848
*Determining in Code Whether a `CheckBox` Is Selected* . . . . . . . . . . 848
*Selecting a `CheckBox` in Code* . . . . . . . . . . 849
*Responding to `CheckBox` Clicks* . . . . . . . . . . 853
13.4 `ListView` Controls . . . . . . . . . . 853
*Retrieving the Selected Item* . . . . . . . . . . 855

*Retrieving the Index of the Selected Item* . . . . . . . . . . . . . . . . . . . . . . . . 856
*Responding to Item Selection with an Event Handler* . . . . . . . . . . . . . . . 859
*Adding Items versus Setting Items* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 860
*Initializing a `ListView` with an Array or an `ArrayList`* . . . . . . . . . . . . 861
*Selection Modes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 862
*Retrieving Multiple Selected Items* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 863
*Working With the Elements of an ObservableList* . . . . . . . . . . . . . . . . . . 866
*Converting an `ObservableList` to an Array* . . . . . . . . . . . . . . . . . . . . . 867
*Using Code to Select an Item in a `ListView`* . . . . . . . . . . . . . . . . . . . . . 868
*`ListView` Orientation* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 868
*Creating ListViews of Objects Other Than `String`* . . . . . . . . . . . . . . . . . 869
13.5 `ComboBox` Controls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .874
*Retrieving the Selected Item* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 875
*Responding to `ComboBox` Item Selection with an Event Handler* . . . . . . 877
*Editable ComboBoxes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 879
13.6 `Slider` Controls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .880
13.7 `TextArea` Controls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .885
13.8 Menus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .887
*Assigning Mnemonics to Menu Items* . . . . . . . . . . . . . . . . . . . . . . . . . . . 895
13.9 The `FileChooser` Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .897
*Displaying a `FileChooser` Dialog Box* . . . . . . . . . . . . . . . . . . . . . . . . . . 898
13.10 Using Console Output to Debug a GUI Application . . . . . . . . . . . . . .898
13.11 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .902
*Review Questions and Exercises* 902
*Programming Challenges* 906

## Chapter 14 **JavaFX: Graphics, Effects, and Media 909**

14.1 Drawing Shapes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .909
*The Screen Coordinate System* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 909
*The `Shape` Class and Its Subclasses* . . . . . . . . . . . . . . . . . . . . . . . . . . 910
*The `Line` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 911
*Changing the Stroke Color* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 914
*The `Circle` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 914
*The `Rectangle` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 918
*The `Ellipse` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 921
*The `Arc` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 924
*The `Polygon` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 928
*The `Polyline` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 931
*The `Text` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 933
*Rotating Nodes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 936
*Scaling Nodes* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 938
14.2 Animation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .940
*The `TranslateTransition` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 941
*The `RotateTransition` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 944
*The `ScaleTransition` Class* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 949

*The `StrokeTransition` Class* . . . . . . . . . . 952
*The `FillTransition` Class* . . . . . . . . . . 953
*The `FadeTransition` Class* . . . . . . . . . . 954
*Controlling the Animation* . . . . . . . . . . 956
*Specifying an Interpolator* . . . . . . . . . . 956
14.3 Effects . . . . . . . . . . 958
*The `DropShadow` Class* . . . . . . . . . . 958
*The `InnerShadow` Class* . . . . . . . . . . 961
*The `ColorAdjust` Class* . . . . . . . . . . 962
*The `BoxBlur`, `GaussianBlur`, and `MotionBlur` Classes* . . . . . . . . . . 962
*The `SepiaTone` Class* . . . . . . . . . . 965
*The `Glow` Class* . . . . . . . . . . 966
*The `Reflection` Class* . . . . . . . . . . 967
*Combining Effects* . . . . . . . . . . 967
14.4 Playing Sound Files . . . . . . . . . . 969
*Registering an `EndOfMedia` Event Handler* . . . . . . . . . . 971
14.5 Playing Videos . . . . . . . . . . 974
14.6 Handling Key Events . . . . . . . . . . 979
*Using an Anonymous Inner Class to Register a Key Event Handler to the Scene* . . . . . . . . . . 980
*Using a Lambda Expression to Register a Key Event Handler to the Scene* . . . . . . . . . . 981
14.7 Handling Mouse Events . . . . . . . . . . 986
14.8 Common Errors to Avoid . . . . . . . . . . 992
*Review Questions and Exercises* 992
*Programming Challenges* 995

## Chapter 15 **Recursion** 999

15.1 Introduction to Recursion . . . . . . . . . . 999
15.2 Solving Problems with Recursion . . . . . . . . . . 1002
*Direct and Indirect Recursion* . . . . . . . . . . 1006
15.3 Examples of Recursive Methods . . . . . . . . . . 1007
*Summing a Range of Array Elements with Recursion* . . . . . . . . . . 1007
*Drawing Concentric Circles* . . . . . . . . . . 1008
*The Fibonacci Series* . . . . . . . . . . 1011
*Finding the Greatest Common Divisor* . . . . . . . . . . 1012
15.4 A Recursive Binary Search Method . . . . . . . . . . 1014
15.5 The Towers of Hanoi . . . . . . . . . . 1017
15.6 Common Errors to Avoid . . . . . . . . . . 1021
*Review Questions and Exercises* 1022
*Programming Challenges* 1025

## Chapter 16 **Sorting, Searching, and Algorithm Analysis** 1027

16.1 Introduction to Sorting Algorithms . . . . . . . . . . 1027
*The Bubble Sort* . . . . . . . . . . 1027

*Using the Bubble Sort to Sort Objects* . . . . . . . . . . . . . . . . . . . . . . . . *1033*
*The Selection Sort.* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1036*
*Using the Selection Sort to Sort Objects* . . . . . . . . . . . . . . . . . . . . . . . *1040*
*The Insertion Sort.* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1042*
*Using the Insertion Sort to Sort Objects* . . . . . . . . . . . . . . . . . . . . . . . *1047*
*The Quicksort Algorithm* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1048*
*Using Quicksort to Sort Objects* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1052*
16.2 Introduction to Search Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . 1052
*The Sequential Search Algorithm* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1052*
*Average Case Analysis of the Sequential Search* . . . . . . . . . . . . . . . . . . *1055*
*The Binary Search* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1055*
*Efficiency of the Binary Search* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1059*
*A Recursive Binary Search* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1059*
16.3 Analysis of Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1063
*Computational Problems and Basic Steps* . . . . . . . . . . . . . . . . . . . . . . . *1064*
*Complexity of Algorithms* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1065*
*Worst Case Complexity of Algorithms* . . . . . . . . . . . . . . . . . . . . . . . . . . *1066*
*Average Case Complexity.* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1067*
*Asymptotic Complexity and the Big O Notation* . . . . . . . . . . . . . . . . . . *1068*
16.4 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1072
*Review Questions and Exercises* 1073
*Programming Challenges* 1076

## Chapter 17 **Generics** **1079**

17.1 Introduction to Generics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1079
*Using the Diamond Operator for Type Inference* . . . . . . . . . . . . . . . . . . *1082*
17.2 Writing a Generic Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1082
*Only Reference Types Can Be Passed to Type Parameters* . . . . . . . . . . . *1087*
*Instantiating a Generic Class without Specifying a Type Argument* . . . *1088*
*Commonly Used Type Parameter Names* . . . . . . . . . . . . . . . . . . . . . . . . *1090*
17.3 Passing Objects of a Generic Class to a Method . . . . . . . . . . . . . . . . 1090
*Constraining a Type Parameter* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1092*
*Defining a Type Parameter in a Method Header* . . . . . . . . . . . . . . . . . . *1094*
*The `extends` Key Word Constrains a Type to an Upper Bound* . . . . . . . *1095*
*The `super` Key Word Constrains a Type to a Lower Bound* . . . . . . . . . . *1095*
17.4 Writing Generic Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1096
17.5 Constraining a Type Parameter in a Generic Class . . . . . . . . . . . . . . . 1097
17.6 Inheritance and Generic Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1100
17.7 Defining Multiple Type Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . 1104
17.8 Generics and Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1107
*Constraining a Type Parameter to a Type That Implements an Interface* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1108*
17.9 Erasure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1113
17.10 Restrictions on the Use of Generic Types . . . . . . . . . . . . . . . . . . . . . 1116
17.11 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1118
*Review Questions and Exercises* 1119
*Programming Challenges* 1122

## Chapter 18 **Collections and the Stream API 1125**

18.1 Introduction to the Java Collections Framework . . . . . . . . . . . . . . . .1125
- *Lists, Sets, and Maps . . . . . . . . . . . . 1125*
- *Java Functional Interfaces . . . . . . . . . . . . 1126*
- *JCF Interfaces . . . . . . . . . . . . 1126*
- *Iterating a Collection . . . . . . . . . . . . 1127*
- *The `Iterable` Interface and the `forEach` method . . . . . . . . . . . . 1127*
- *Iterators . . . . . . . . . . . . 1128*
- *The Enhanced `For` Loop . . . . . . . . . . . . 1131*
- *The `Collection` Interface . . . . . . . . . . . . 1131*

18.2 Lists . . . . . . . . . . . . 1133
- *The `List` Interface . . . . . . . . . . . . 1134*
- *The `ArrayList` and `LinkedList` Classes . . . . . . . . . . . . 1135*
- *Creating and Initializing Lists . . . . . . . . . . . . 1136*
- *List Iterators . . . . . . . . . . . . 1137*
- *Methods of the `LinkedList` Class . . . . . . . . . . . . 1140*
- *Using an Interface Variable to Reference a `Collection` Object . . . . . . . 1141*

18.3 Sets . . . . . . . . . . . . 1143
- *The `Set` Interface . . . . . . . . . . . . 1143*
- *The `HashSet` Class . . . . . . . . . . . . 1143*
- *Implementing Your Own `hashCode` Method . . . . . . . . . . . . 1148*
- *The `LinkedHashSet` Class . . . . . . . . . . . . 1152*
- *The `SortedSet` Interface and the `TreeSet` Class . . . . . . . . . . . . 1153*
- *Using a Comparator to Compare Objects . . . . . . . . . . . . 1156*

18.4 Maps . . . . . . . . . . . . 1158
- *The `HashMap` Class . . . . . . . . . . . . 1160*
- *The `LinkedHashMap` Class . . . . . . . . . . . . 1166*
- *The `SortedMap` Interface and the `TreeMap` Class . . . . . . . . . . . . 1168*

18.5 The `Collections` Class . . . . . . . . . . . . 1169

18.6 The Stream API . . . . . . . . . . . . 1171
- *The Stream Package . . . . . . . . . . . . 1172*
- *Creating a Stream . . . . . . . . . . . . 1172*
- *Terminal Operations . . . . . . . . . . . . 1173*
- *Intermediate Operations on Streams . . . . . . . . . . . . 1177*
- *Intermediate Stream Operations are Lazy . . . . . . . . . . . . 1179*
- *The Stream `map` Operation . . . . . . . . . . . . 1181*
- *The Stream `reduce` Methods . . . . . . . . . . . . 1182*
- *Stream Collectors . . . . . . . . . . . . 1184*

18.7 Common Errors to Avoid . . . . . . . . . . . . 1189

*Review Questions and Exercises* 1190

*Programming Challenges* 1193

## Chapter 19 **Linked Lists 1195**

19.1 Introduction to Linked Lists . . . . . . . . . . . . 1195
- *Creating Lists and Adding Nodes . . . . . . . . . . . . 1196*

*Removing a Node* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1198*
*Traversing Linked Lists* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1199*
19.2 Operations on Linked Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1202
*The `isEmpty` Method* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1202*
*The `size` Method* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1203*
*The `add` Methods* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1203*
*The `remove` Methods* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1204*
*The `toString` Method* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1204*
*A Graphical Interface for the Linked List Program* . . . . . . . . . . . . . . . . *1210*
19.3 Doubly Linked and Circularly Linked Lists . . . . . . . . . . . . . . . . . . . . . .1215
*Adding a Node to a Doubly Linked List* . . . . . . . . . . . . . . . . . . . . . . . . *1216*
*Removing a Node from a Doubly Linked List* . . . . . . . . . . . . . . . . . . . . *1217*
*Circularly Linked Lists* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1223*
19.4 Recursion on Linked Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1224
*A Recursive Implementation of `size`* . . . . . . . . . . . . . . . . . . . . . . . . . . *1224*
*A Recursive Implementation of the `add(String e, Node list)` Method* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1225*
*A Recursive Implementation of `add(int index, String e, Node list)`* . .*1226*
*A Recursive Implementation of the `remove` Methods* . . . . . . . . . . . . . . *1226*
19.5 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1238
*Review Questions and Exercises* 1238
*Programming Challenges* 1241

## Chapter 20 **Stacks and Queues 1245**

20.1 Stacks and Their Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1245
*Examples and Applications of Stacks* . . . . . . . . . . . . . . . . . . . . . . . . . . *1245*
*Stack Operations* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1246*
*Stacks of Primitive Types* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1247*
20.2 Array Implementation of Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1248
*The Stack Push Operation* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1249*
*The Stack `empty` Method* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1249*
*The Stack `peek` and `pop` Methods* . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1249*
*Stacks of Objects* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1253*
20.3 Linked Implementation of Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1254
*Implementation of Stack Methods* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1255*
20.4 Queues and Their Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1259
20.5 Array Implementation of Queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1259
*GUI Front End for the Queue Demo Program* . . . . . . . . . . . . . . . . . . . . *1266*
20.6 Linked List Implementation of Queues . . . . . . . . . . . . . . . . . . . . . . . .1270
*Queue Initialization and Enqueuing of Items* . . . . . . . . . . . . . . . . . . . *1270*
*Dequeuing Items* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *1271*
20.7 Generic Implementation of Stacks and Queues . . . . . . . . . . . . . . . . .1275
20.8 Queues and Breadth-First Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1278
20.9 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1281
*Review Questions and Exercises* 1282
*Programming Challenges* 1284

## Chapter 21 **Binary Trees, AVL Trees, and Priority Queues 1287**

21.1 Binary Trees and Their Applications . . . . . . . . . . . . . . . . . . . . . . . . . .1287

*Binary Tree Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1288*

*Applications of Binary Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1289*

*Representing Nodes of Binary Trees . . . . . . . . . . . . . . . . . . . . . . . . . 1289*

*Traversing a Binary Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1290*

*Graphical Display of Binary Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . 1293*

*Class Implementation of Binary Trees . . . . . . . . . . . . . . . . . . . . . . . . 1297*

21.2 Binary Search Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1297

*Adding a Value to a Binary Search Tree . . . . . . . . . . . . . . . . . . . . . . 1299*

*Removing a Value from a Binary Search Tree . . . . . . . . . . . . . . . . . . . 1299*

*Implementation of Binary Search Trees . . . . . . . . . . . . . . . . . . . . . . . 1303*

*Graphical Display of Binary Search Trees . . . . . . . . . . . . . . . . . . . . . . 1308*

*A Program for Testing the Binary Search Tree Class . . . . . . . . . . . . . . . 1310*

21.3 AVL Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1315

*Adding New Elements to AVL Trees . . . . . . . . . . . . . . . . . . . . . . . . . 1316*

*Implementation of AVL Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1319*

21.4 Priority Queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1328

*Applications of Priority Queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1330*

*Using Comparators with Priority Queues . . . . . . . . . . . . . . . . . . . . . . 1331*

*Analysis of Heapsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1333*

*Implementation of Priority Queues . . . . . . . . . . . . . . . . . . . . . . . . . . 1333*

*Binary Trees with the Heap Order Property . . . . . . . . . . . . . . . . . . . . 1333*

*Complete Binary Trees and Heaps . . . . . . . . . . . . . . . . . . . . . . . . . . . 1334*

*The Depth of Complete Binary Trees . . . . . . . . . . . . . . . . . . . . . . . . . 1335*

*Storing a Complete Binary Tree in an Array . . . . . . . . . . . . . . . . . . . . 1335*

*Adding an Item to a Heap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1336*

*Removing the Minimum Element. . . . . . . . . . . . . . . . . . . . . . . . . . . . 1337*

21.5 Common Errors to Avoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1345

*Review Questions and Exercises* 1345

*Programming Challenges* 1349

**Index 1353**

The following appendices, online chapters, and online case studies are available on the book's online resource page at www.pearson.com/gaddis.

## Online Appendices

Appendix A: The ASCII/Unicode Characters
Appendix B: Operator Precedence and Associativity
Appendix C: Java Key Words
Appendix D: Installing the JDK and Using the JDK Documentation
Appendix E: Using the `javadoc` Utility
Appendix F: More about the Math Class
Appendix G: Packages
Appendix H: Working with Records and Random Access Files
Appendix I: Configuring JavaDB and Installing Apache Derby
Appendix J: The QuickSort Algorithm
Appendix K: Named Colors
Appendix L: Answers to Checkpoints
Appendix M: Answers to Odd-Numbered Review Questions

## Online Chapters

Chapter 22: Databases
Chapter 23: A First Look at GUI Applications with Swing
Chapter 24: Advanced Swing GUI Applications
Chapter 25: Applets and More

## Online Case Studies

Case Study 1: Calculating Sales Commission
Case Study 2: The `Amortization` Class
Case Study 3: The `PinTester` Class
Case Study 4: Parallel Arrays
Case Study 5: The `FeetInches` Class
Case Study 6: The `SerialNumber` Class
Case Study 7: A Simple Text Editor Application

## LOCATION OF VIDEONOTES IN THE TEXT

| | |
|---|---|
| **Chapter 1** | Compiling and Running a Java Program, p. 14<br>Using an IDE, p. 14<br>Your First Java Program, p. 25 |
| **Chapter 2** | Displaying Console Output, p. 33<br>Declaring Variables, p. 39<br>Simple Math Expressions, p. 55<br>The Miles-per-Gallon Problem, p. 107 |
| **Chapter 3** | The `if` Statement, p. 111<br>The `if-else` Statement, p. 120<br>The `if-else-if` Statement, p. 129<br>The Time Calculator Problem, p. 182 |
| **Chapter 4** | The while Loop, p. 193<br>The Pennies for Pay Problem, p. 263 |
| **Chapter 5** | Passing Arguments to a Method, p. 279<br>Returning a Value from a Method, p. 293<br>The Retail Price Calculator Problem, p. 311 |
| **Chapter 6** | Writing Classes and Creating Objects, p. 325<br>Initializing an Object with a Constructor, p. 346<br>The Personal Information Class Problem, p. 395 |
| **Chapter 7** | Accessing Array Elements in a Loop, p. 407<br>Passing an Array to a Method, p. 422<br>The Charge Account Validation Problem, p. 487 |
| **Chapter 8** | Returning Objects from Methods, p. 503<br>Aggregation, p. 515<br>The `BankAccount`, Class Copy Constructor Problem, p. 552 |
| **Chapter 9** | The Sentence Capitalizer Problem, p. 605 |
| **Chapter 10** | Inheritance, p. 611<br>Polymorphism, p. 655<br>The `Employee` and `Productionworker` Classes Problem, p. 696 |
| **Chapter 11** | Handling Exceptions, p. 701<br>The Exception Project Problem, p. 757 |
| **Chapter 12** | Introduction to JavaFX, p. 762<br>Creating Scenes, p. 765<br>Displaying Images, p. 772<br>The HBox Layout Container, p. 777<br>The VBox Layout Container, p. 782<br>The `GridPane` Layout Container, p. 784<br>`Button` Controls and Events, p. 792<br>The TextField Control, p. 799<br>Using Anonymous Inner Classes as Event Handlers, p. 803<br>Using Lambda Expressions as Event Handlers, p. 806<br>The Latin Translator Problem, p. 818 |

(*continued on the next page*)

## LOCATION OF VIDEONOTES IN THE TEXT *(continued)*

| | |
|---|---|
| **Chapter 13** | JavaFX and CSS, p. 823<br>`RadioButton` Controls, p. 838<br>`CheckBox` Controls, p. 848<br>`ListView` Controls, p. 853<br>`ComboBox` Controls, p. 874<br>`Slider` Controls, p. 880<br>The Dorm and Meal Plan Calculator Problem, p. 906 |
| **Chapter 14** | Drawing Shapes with JavaFX, p. 909<br>JavaFX Animation, p. 940<br>JavaFX Effects, p. 958<br>Playing Sound Files with JavaFX, p. 970<br>Playing Videos with JavaFX, p. 974<br>Handling Key Events in JavaFX, p. 979<br>Handling Mouse Events in JavaFX, p. 986<br>The Coin Toss Problem, p. 996 |
| **Chapter 15** | Reducing a Problem with Recursion, p. 1003<br>The Recursive Power Problem, p. 1026 |
| **Chapter 16** | The Bubble Sort Algorithm, p. 1027<br>Searching for Objects with the Binary Search Algorithm, p. 1076 |
| **Chapter 17** | Inheritance and Generic classes, p. 1100<br>Highest and Lowest Elements, p. 1123 |
| **Chapter 18** | The `HashMap` Class, p. 1160 |
| **Chapter 19** | Adding a Node to a Doubly-Linked List, p. 1216<br>Recursive Linked List Operations, p. 1242 |
| **Chapter 20** | Array Implementation of Queues, p. 1259<br>Array-Based Deque, p. 1285 |
| **Chapter 21** | Removing a Value from a Binary Search Tree, p. 1299<br>Prefix Calculator, p. 1350 |

# Preface

Welcome to *Starting Out with Java: From Control Structures through Data Structures, Fourth Edition*. This book is intended for a traditional two-semester CS1/CS2 sequence of courses. The first half of the book, intended for a CS1 class, teaches fundamental programming and problem-solving concepts using Java. The second half of the book, intended for a CS2 class, teaches advanced concepts and provides an introduction to algorithms and data structures. The book is written for students with no prior programming background, but experienced students will also benefit from its depth of detail.

## Control Structures First, Then Objects, Then Data Structures

The text introduces students to the fundamentals of data types, input/output, control structures, methods, and objects created from standard library classes. Next, students learn to use arrays. Then, a series of more advanced topics are presented, including inheritance, polymorphism, GUI development, recursion, searching, sorting, algorithm analysis, and generics. Finally, students are introduced to the world of data structures, beginning with an overview of the Java Collections Framework (JCF), and continuing through a series of chapters in which students learn to develop their own data structures, including linked lists, stacks, queues, priority queues, binary trees, and AVL trees. Students are shown how to implement data structures either with or without generics, which gives instructors a great deal of flexibility in presenting the material.

As with all the books in the *Starting Out With* series, the hallmark of the text is its clear, friendly, and easy-to-understand writing. In addition, it is rich in concise, practical example programs.

## Changes in This Edition

This book's pedagogy, organization, and clear writing style remain the same as in the previous edition. The most significant change in this edition is the switch from Swing to JavaFX in the chapters that focus on GUI development. Although Swing is not officially deprecated, Oracle has announced that JavaFX has replaced Swing as the standard GUI library for Java[1].

---

[1] http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6

In this edition, we have added the following new chapters:

- **Chapter 12 JavaFX: GUI Programming and Basic Controls:** This chapter presents the basics of developing graphical user interface (GUI) applications with JavaFX. Fundamental controls, layout containers, and the basic concepts of event-driven programming are covered.
- **Chapter 13 JavaFX: Advanced Controls:** This chapter discusses CSS styling and advanced user interface controls.
- **Chapter 14 JavaFX: Graphics, Effects, and Media:** This chapter discusses 2D shapes, animation, visual effects, playing audio and video, and responding to mouse and keyboard events.

The Swing and Applet material that appeared in the previous edition is still available on the book's companion Web site, as the following online chapters:

- The previous Chapter 12 *A First Look At GUI Applications* is now available online as Chapter 23.
- The previous Chapter 13 *Advanced GUI Applications* is now available online as Chapter 24.
- The previous Chapter 14 *Applets and More* is now available online as Chapter 25.

In addition to the new JavaFX chapters, the online Database chapter, which is now Chapter 22, has been updated to use JavaFX instead of Swing for its GUI applications. We have also added several new, motivational programming problems throughout the book.

## Organization of the Text

The text teaches Java step by step. Each chapter covers a major set of topics and builds knowledge as students progress through the book. Although the chapters can be easily taught in their existing sequence, there is some flexibility. Figure P-1 shows chapter dependencies. Each box represents a chapter or a group of chapters. An arrow points from a chapter to the chapter that must be previously covered.

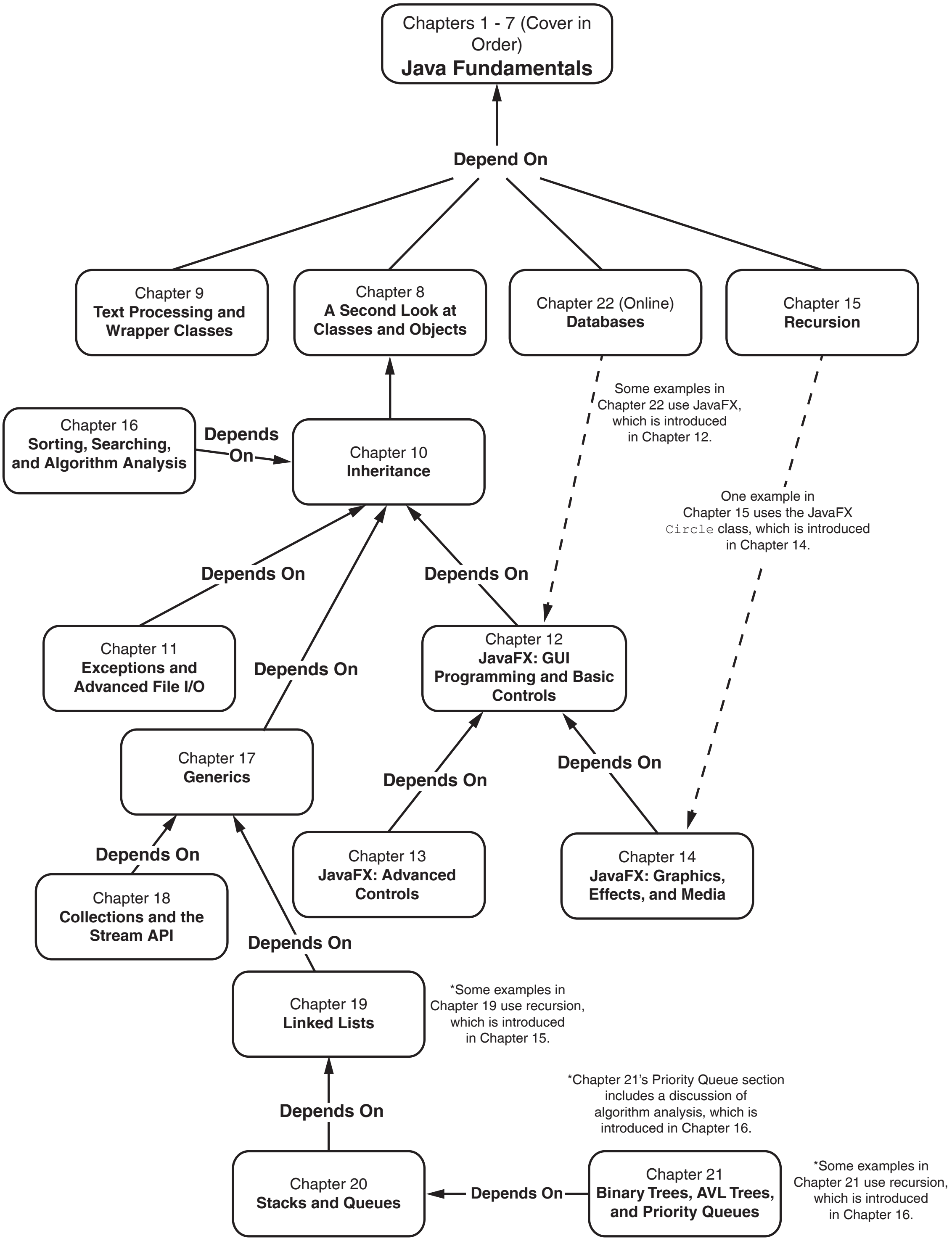
## Brief Overview of Each Chapter

**Chapter 1: Introduction to Computers and Java.** This chapter provides an introduction to the field of computer science, and covers the fundamentals of hardware, software, and programming languages. The elements of a program, such as key words, variables, operators, and punctuation are discussed by examining a simple program. An overview of entering source code, compiling, and executing a program is presented. A brief history of Java is also given.

**Chapter 2: Java Fundamentals.** This chapter gets students started in Java by introducing data types, identifiers, variable declarations, constants, comments, program output, and simple arithmetic operations. The conventions of programming style are also introduced. Students learn to read console input with the Scanner class and with dialog boxes using `JOptionPane`.

**Chapter 3: Decision Structures.** In this chapter students explore relational operators and relational expressions, and are shown how to control the flow of a program with the `if`,

**Figure P-1** Chapter dependencies


Chapters 1 - 7 (Cover in Order)
Java Fundamentals
Depend On
Chapter 9
Text Processing and Wrapper Classes
Chapter 8
A Second Look at Classes and Objects
Chapter 22 (Online)
Databases
Chapter 15
Recursion
Chapter 16
Sorting, Searching, and Algorithm Analysis
Depends On
Chapter 10
Inheritance
Some examples in Chapter 22 use JavaFX, which is introduced in Chapter 12.
One example in Chapter 15 uses the JavaFX Circle class, which is introduced in Chapter 14.
Depends On
Depends On
Chapter 11
Exceptions and Advanced File I/O
Depends On
Chapter 12
JavaFX: GUI Programming and Basic Controls
Chapter 17
Generics
Depends On
Depends On
Chapter 13
JavaFX: Advanced Controls
Chapter 14
JavaFX: Graphics, Effects, and Media
Depends On
Chapter 18
Collections and the Stream API
Depends On
Chapter 19
Linked Lists
*Some examples in Chapter 19 use recursion, which is introduced in Chapter 15.
Depends On
*Chapter 21's Priority Queue section includes a discussion of algorithm analysis, which is introduced in Chapter 16.
Chapter 20
Stacks and Queues
Depends On
Chapter 21
Binary Trees, AVL Trees, and Priority Queues
*Some examples in Chapter 21 use recursion, which is introduced in Chapter 16.

`if-else`, and `if-else-if` statements. Nested `if` statements, logical operators, the conditional operator, and the `switch` statement are also covered. The chapter discusses how to compare `String` objects with the `equals`, `compareTo`, `equalsIgnoreCase`, and `compareToIgnoreCase` methods. Formatting numeric output with the `System.out.printf` method and the `String.format` method is discussed.

**Chapter 4: Loops and Files.** This chapter covers Java's repetition control structures. The `while loop`, `do-while loop`, and `for loop` are taught, along with common uses for these devices. Counters, accumulators, running totals, sentinels, and other application-related topics are discussed. Simple file operations for reading and writing text files are included.

**Chapter 5: Methods.** In this chapter students learn how to write `void` methods, value-returning methods, and methods that do and do not accept arguments. The concept of functional decomposition is discussed.

**Chapter 6: A First Look at Classes.** This chapter introduces students to designing classes for the purpose of instantiating objects. Students learn about class fields and methods, and UML diagrams are introduced as a design tool. Then constructors and overloading are discussed. A `BankAccount` class is presented as a case study, and a section on object-oriented design is included. This section leads the students through the process of identifying classes and their responsibilities within a problem domain. There is also a section that briefly explains packages and the `import` statement.

**Chapter 7: Arrays and the `ArrayList` Class.** In this chapter students learn to create and work with single and multi-dimensional arrays. Numerous array-processing techniques are demonstrated, such as summing the elements in an array, finding the highest and lowest values, and sequentially searching an array are also discussed. Other topics, including ragged arrays and variable-length arguments (varargs) are also discussed. The `ArrayList` class is introduced and Java's generic types are briefly discussed and demonstrated.

**Chapter 8: A Second Look at Classes and Objects.** This chapter shows students how to write classes with added capabilities. Static methods and fields, interaction between objects, passing objects as arguments, and returning objects from methods are discussed. Aggregation and the "has a" relationship is covered, as well as enumerated types. A section on object-oriented design shows how to use CRC cards to determine the collaborations among classes.

**Chapter 9: Text Processing and More about Wrapper Classes.** This chapter discusses the numeric and `Character` wrapper classes. Methods for converting numbers to strings, testing the case of characters, and converting the case of characters are covered. Autoboxing and unboxing are also discussed. More `String` class methods are covered, including using the `split` method to tokenize strings. The chapter also covers the `StringBuilder` class.

**Chapter 10: Inheritance.** The study of classes continues in this chapter with the subjects of inheritance and polymorphism. The topics covered include superclasses, subclasses, how constructors work in inheritance, method overriding, polymorphism and dynamic binding, protected and package access, class hierarchies, abstract classes, abstract methods, anonymous inner classes, interfaces, and lambda expressions.

**Chapter 11: Exceptions and Advanced File I/O.** In this chapter students learn to develop enhanced error trapping techniques using exceptions. Handling exceptions is covered, as well as developing and throwing custom exceptions. The chapter discusses advanced techniques for working with sequential access, random access, text, and binary files.

**Chapter 12: JavaFX: GUI Programming and Basic Controls.** This chapter presents the basics of developing graphical user interface (GUI) applications with JavaFX. Fundamental controls, layout containers, and the basic concepts of event-driven programming are covered.

**Chapter 13: JavaFX: Advanced Controls.** This chapter discusses CSS styling and advanced user interface controls, such as RadioButtons, CheckBoxes, ListViews, ComboBoxes, Sliders, and TextAreas. Menu systems and FileChoosers are also covered.

**Chapter 14: JavaFX: Graphics, Effects, and Media.** This chapter discusses 2D shapes, animation, visual effects, playing audio and video, and responding to mouse and keyboard events.

**Chapter 15: Recursion.** This chapter presents recursion as a problem-solving technique. Numerous examples of recursive methods are demonstrated.

**Chapter 16: Sorting, Searching, and Algorithm Analysis.** In this chapter students learn the basics of sorting arrays and searching for data stored in them. The chapter covers the bubble sort, selection sort, insertion sort, Quicksort, sequential search, and binary search algorithms. A section on algorithm analysis is also provided, which covers basic steps, complexity, and Big O notation.

**Chapter 17: Generics.** This chapter shows students how to write generic classes and methods. Topics include erasure, passing instances of a generic class as arguments to a method, constraining type parameters, generics and inheritance, and generics and interfaces. Restrictions on the use of generic types are also discussed.

**Chapter 18: Collections and the Stream API.** This chapter introduces students to the Java Collections Framework (JCF). Lists, sets, maps, and the Collections class are discussed. The chapter concludes with a discussion of the classes and interfaces in the java.util.stream package.

**Chapter 19: Linked Lists.** This chapter introduces concepts and techniques for writing a linked list class. Basic linked list operations, such as adding, inserting, removing, and traversing, are covered. Doubly-linked lists, circularly-linked lists, and using recursion with linked lists are discussed.

**Chapter 20: Stacks and Queues.** In this chapter students learn about the operations of stacks and queues, and how to write array-based and linked list-based stack and queue classes.

**Chapter 21: Binary Trees, AVL Trees, and Priority Queues.** This chapter covers various aspects of binary trees and binary tree operations. The specific binary tree implementations covered are binary search trees, AVL trees, and priority queues.

**Chapter 22 (Online): Databases.** This chapter is available on the book's companion Web site (at www.pearson.com/gaddis). It introduces the students to database programming. The basic concepts of database management systems and SQL are first introduced. Then the students learn to use JDBC to write database applications in Java. Relational data is covered, and numerous example programs are presented throughout the chapter.

**Chapters 23 – 25 (Online): Legacy Swing GUI Chapters.** The previous edition's chapters on Swing GUI and Applet programming are now available on the book's companion website (at www.pearson.com/gaddis) as Chapters 23 through 25.

## Features of the Text

**Concept Statements.** Each major section of the text starts with a concept statement that concisely summarizes the focus of the section.

**Example Programs.** The text has an abundant number of complete and partial example programs, each designed to highlight the current topic. In most cases the programs are practical, real-world examples.

**Program Output.** Each example program is followed by a sample of its output, which shows students how the program functions.

**Checkpoints.** Checkpoints, highlighted by the checkmark icon, appear at intervals throughout each chapter. They are designed to check students' knowledge soon after learning a new topic. Answers for all Checkpoint questions can be downloaded from the book's companion Web site at www.pearson.com/gaddis.

**NOTE:** Notes appear at several places throughout the text. They are short explanationsof interesting or often misunderstood points relevant to the topic at hand.

**TIP:** Tips advise the student on the best techniques for approaching different programming problems and appear regularly throughout the text.

**WARNING!** Warnings caution students about certain Java features, programming techniques, or practices that can lead to malfunctioning programs or lost data.

**In the Spotlight.** Many of the chapters provide an *In the Spotlight* section that presents a programming problem, along with detailed, step-by-step analysis showing the student how to solve it.

VideoNote

**VideoNotes.** A series of online videos, developed specifically for this book, are available for viewing at www.pearson.com/gaddis. Icons appear throughout the text, alerting the student to videos about specific topics.

**Case Studies.** Case studies that simulate real-world business applications are introduced throughout the text and are available for download from the Gaddis resource page at www.pearson.com/gaddis.

**Common Errors to Avoid.** Most chapters provide a list of common errors and explanations of how to avoid them.

**Review Questions and Exercises.** Each chapter presents a thorough and diverse set of review questions and exercises. They include Multiple Choice and True/False, Find the Error, Algorithm Workbench, and Short Answer.

**Programming Challenges.** Each chapter offers a pool of programming challenges designed to solidify students' knowledge of topics at hand. In most cases the assignments present real-world problems to be solved.

## Supplements

### Student Online Resources

Many student resources are available for this book from the publisher. The following items are available on the Gaddis Series resource page at www.pearson.com/gaddis:

- The source code for each example program in the book
- Access to the book's companion VideoNotes
- Appendixes A–M (listed in the Contents)
- A collection of seven valuable Case Studies (listed in the Contents)
- Links to download the Java™ Edition Development Kit
- Links to download numerous programming environments, including jGRASP™, Eclipse™, TextPad™, NetBeans™, JCreator, and DrJava

### Instructor Resources

The following supplements are available to qualified instructors:

- Answers to all of the Review Questions
- Solutions for the Programming Challenges
- PowerPoint presentation slides for each chapter

## Acknowledgments

There have been many helping hands in the development and publication of this book. We would like to thank the following faculty reviewers for their helpful suggestions and expertise:

John Bono
*George Mason University*

Irene Bruno
*George Mason University*

Richard Burns
*Westchester University*

Jackie Horton
*University of Vermont*

Mohammad T. Islam
*Southern Connecticut State University*

David Krebs
*University of Pittsburgh*

Michael Ruth
*Roosevelt University*

Christian Servin
*El Paso Community College*

Li Yang
*Western Michigan University*

### Reviewers for Previous Editions

Ahmad Abuhejleh
*University of Wisconsin, River Falls*

Colin Archibald
*Valencia Community College*

Ijaz Awani
*Savannah State University*

Bill Bane
*Tarleton State University*

N. Dwight Barnette
*Virginia Tech*

Asoke Bhattacharyya
*Saint Xavier University, Chicago*

Marvin Bishop
*Manhattan College*

eather Booth
*University of Tennessee, Knoxville*

David Boyd
*Valdosta State University*

Julius Brandstatter
*Golden Gate University*

Kim Cannon
*Greenville Tech*

Jesse Cecil
*College of the Siskiyous*

James Chegwidden
*Tarrant County College*

Kay Chen
*Bucks County Community College*

Brad Chilton
*Tarleton State University*

Diane Christie
*University of Wisconsin, Stout*

Cara Cocking
*Marquette University*

Jose Cordova
*University of Louisiana, Monroe*

Walter C. Daugherity
*Texas A&M University*

Michael Doherty
*University of the Pacific*

Jeanne M. Douglas
*University of Vermont*

Sander Eller
*California Polytechnic University, Pomona*

Brooke Estabrook-Fishinghawk
*Mesa Community College*

Mike Fry
*Lebanon Valley College*

David Goldschmidt
*College of St. Rose*

Georgia R. Grant
*College of San Mateo*

Carl Stephen Guynes
*University of North Texas*

Nancy Harris
*James Madison University*

Chris Haynes
*Indiana University*

Ric Heishman
*Northern Virginia Community College*

Deedee Herrera
*Dodge City Community College*

Mary Hovik
*Lehigh Carbon Community College*

Brian Howard
*DePauw University*

Alan Jackson
*Oakland Community College (MI)*

Norm Jacobson
*University of California, Irvine*

Zhen Jiang
*West Chester University*

Stephen Judd
*University of Pennsylvania*

Neven Jurkovic
*Palo Alto College*

Dennis Lang
*Kansas State University*

Jiang Li
*Austin Peay State University*

Harry Lichtbach
*Evergreen Valley College*

Michael A. Long
*California State University, Chico*

Cheng Luo
*Coppin State University*

Tim Margush
*University of Akron*

Blayne E. Mayfield
*Oklahoma State University*

Scott McLeod
*Riverside Community College*

Dean Mellas
*Cerritos College*

Georges Merx
*San Diego Mesa College*

Martin Meyers
*California State University, Sacramento*

Pati Milligan
*Baylor University*

Laurie Murphy
*Pacific Lutheran University*

Steve Newberry
*Tarleton State University*

Lynne O'Hanlon
*Los Angeles Pierce College*

Merrill Parker
*Chattanooga State Technical Community College*

Bryson R. Payne
*North Georgia College and State University*

Rodney Pearson
*Mississippi State University*

Peter John Polito
*Springfield College*

Charles Robert Putnam
*California State University, Northridge*

Y. B. Reddy
*Grambling State University*

Elizabeth Riley
*Macon State College*

Felix Rodriguez
*Naugatuck Valley Community College*

Diane Rudolph
*John A Logan College*

Carolyn Schauble
*Colorado State University*

Bonnie Smith
*Fresno City College*

Daniel Spiegel
*Kutztown University*

Caroline St. Clair
*North Central College*

Karen Stanton
*Los Medanos College*

Peter van der Goes
*Rose State College*

Timothy Urness
*Drake University*

Tuan A Vo
*Mt. San Antonio College*

Xiaoying Wang
*University of Mississippi*

Yu Wu
*University of North Texas*

Zijiang Yang
*Western Michigan University*

The authors would like to thank their families for their tremendous support during the development of this book. We also want to thank everyone at Pearson for making the *Starting Out With* series so successful. We are extremely fortunate to have Matt Goldstein as our editor and Meghan Jacoby as editorial assistant. Their support and encouragement makes it a pleasure to write chapters and meet deadlines. We are also fortunate to have Demetrius Hall as our marketing manager. He consistently does a great job getting this book out to the academic community. We had a great production team, led by Amanda Brands. She worked tirelessly to make this edition a reality. Thanks to you all!

## About the Authors

**Tony Gaddis** is the principal author of the *Starting Out With* series of textbooks. Tony has nearly 20 years experience teaching computer science courses at Haywood Community College in North Carolina. He is a highly acclaimed instructor who was previously selected as the North Carolina Community College Teacher of the Year and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. The *Starting Out With* series includes introductory books using the C++ programming language, the Java™ programming language, Microsoft® Visual Basic®, Microsoft® C#®, Python, Programming Logic and Design, MIT App Inventor, and Alice, all published by Pearson.

**Godfrey Muganda** is Professor of Computer Science at North Central College. He teaches a wide variety of courses at the undergraduate and graduate levels including courses in Linux and Unix programming, Windows and .NET programming, web application development, web services, data structures, and algorithms. He is a past winner of the North Central College faculty award for outstanding scholarship. His primary research interests are in the area of fuzzy sets and systems.

CHAPTER

# 1

# Introduction to Computers and Java

## TOPICS

- 1.1 Introduction
- 1.2 Why Program?
- 1.3 Computer Systems: Hardware and Software
- 1.4 Programming Languages
- 1.5 What Is a Program Made Of?
- 1.6 The Programming Process
- 1.7 Object-Oriented Programming

## 1.1 Introduction

This book teaches programming using Java. Java is a powerful language that runs on practically every type of computer. It can be used to create large applications, small programs, mobile applications, and code that powers a Web site. Before plunging right into learning Java, however, this chapter will review the fundamentals of computer hardware and software, and then take a broad look at computer programming in general.

## 1.2 Why Program?

**CONCEPT: Computers can do many different jobs because they are programmable.**

Every profession has tools that make the job easier to do. Carpenters use hammers, saws, and measuring tapes. Mechanics use wrenches, screwdrivers, and ratchets. Electronics technicians use probes, scopes, and meters. Some tools are unique and can be categorized as belonging to a single profession. For example, surgeons have certain tools that are designed specifically for surgical operations. Those tools probably aren't used by anyone other than surgeons. There are some tools, however, that are used in several professions. Screwdrivers, for instance, are used by mechanics, carpenters, and many others.

The computer is a tool used by so many professions that it cannot be easily categorized. It can perform so many different jobs that it is perhaps the most versatile tool ever made. To the accountant, computers balance books, analyze profits and losses, and prepare tax reports. To the factory worker, computers control manufacturing machines and track production. To the mechanic, computers analyze the various systems in an automobile and pinpoint hard-to-find problems. The computer can do such a wide variety of tasks because it can be *programmed*. It is a machine specifically designed to follow instructions. Because of the computer's programmability, it doesn't belong to any single profession. Computers are designed to do whatever job their programs, or *software*, tell them to do.

Computer programmers do a very important job. They create software that transforms computers into the specialized tools of many trades. Without programmers, the users of computers would have no software, and without software, computers would not be able to do anything.

Computer programming is both an art and a science. It is an art because every aspect of a program should be carefully designed. Here are a few of the things that must be designed for any real-world computer program:

- The logical flow of the instructions
- The mathematical procedures
- The layout of the programming statements
- The appearance of the screens
- The way information is presented to the user
- The program's "user friendliness"
- Manuals, help systems, and/or other forms of written documentation

There is also a science to programming. Because programs rarely work right the first time they are written, a lot of analyzing, experimenting, correcting, and redesigning is required. This demands patience and persistence of the programmer. Writing software demands discipline as well. Programmers must learn special languages such as Java because computers do not understand English or other human languages. Programming languages have strict rules that must be carefully followed.

Both the artistic and scientific nature of programming makes writing computer software like designing a car: Both cars and programs should be functional, efficient, powerful, easy to use, and pleasing to look at.

## 1.3 Computer Systems: Hardware and Software

**CONCEPT: All computer systems consist of similar hardware devices and software components.**

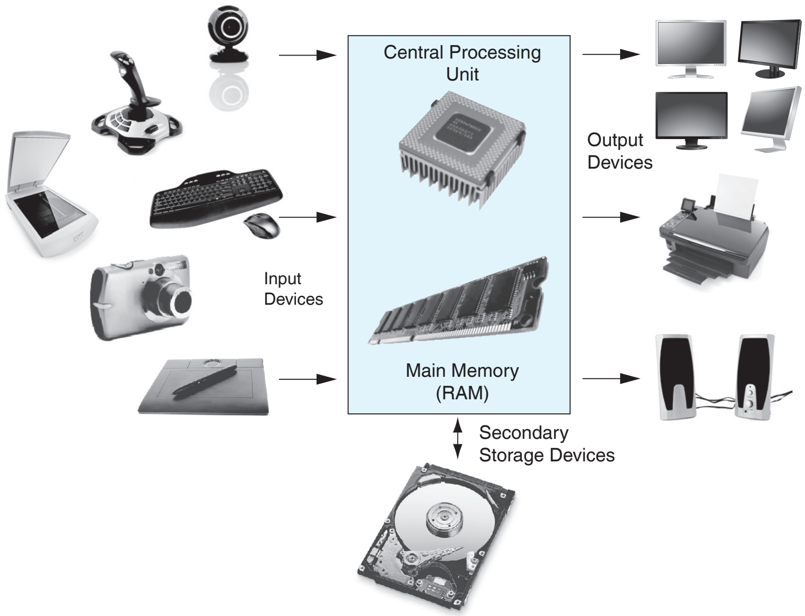
### Hardware

*Hardware* refers to the physical components that a computer is made of. A computer, as we generally think of it, is not an individual device, but a system of devices. Like the instruments in a symphony orchestra, each device plays its own part. A typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

The organization of a computer system is shown in Figure 1-1.

**Figure 1-1** The organization of a computer system 1-1a (TL) iko/Shutterstock, 1-1b (TL) Nikita Rogul/Shutterstock, 1-1c (TLC) Feng Yu/Shutterstock , 1-1d (TLC) Chiyacat/Shutterstock, 1-1e Elkostas/Shutterstock, 1-1f (TLB) tkemot/Shutterstock, 1-1g (BC) Vitaly Korovin/Shutterstock, 1-1h (TRB) Lusoimages/Shutterstock, 1-1i (TRC) jocic/Shutterstock, 1-1j (TR) Best Pictures here/Shutterstock, 1-1k (MC) Peter Guess/Shutterstock, 1-1l (TC) Aquila/Shutterstock

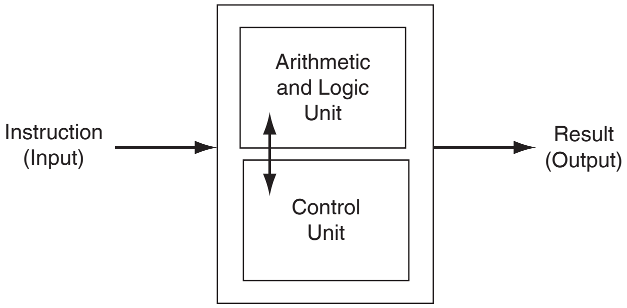

Central Processing Unit
Output Devices
Input Devices
Main Memory (RAM)
Secondary Storage Devices

Let's take a closer look at each of these devices.

### The CPU

At the heart of a computer is its *central processing unit*, or *CPU*. The CPU's job is to fetch instructions, follow the instructions, and produce some resulting data. Internally, the central processing unit consists of two parts: the *control unit* and the *arithmetic and logic unit (ALU)*. The control unit coordinates all of the computer's operations. It is responsible for determining where to get the next instruction and regulating the other major components of the computer with control signals. The arithmetic and logic unit, as its name suggests, is designed to perform mathematical operations. The organization of the CPU is shown in Figure 1-2.

**Figure 1-2** The organization of the CPU


Arithmetic and Logic Unit
Instruction (Input)
Result (Output)
Control Unit

A program is a sequence of instructions stored in the computer's memory. When a computer is running a program, the CPU is engaged in a process known formally as the *fetch/decode/execute cycle*. The steps in the fetch/decode/execute cycle are as follows:

| | |
|---|---|
| *Fetch* | The CPU's control unit fetches, from main memory, the next instruction in the sequence of program instructions. |
| *Decode* | The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal. |
| *Execute* | The signal is routed to the appropriate component of the computer (such as the ALU, a disk drive, or some other device). The signal causes the component to perform an operation. |

These steps are repeated as long as there are instructions to perform.

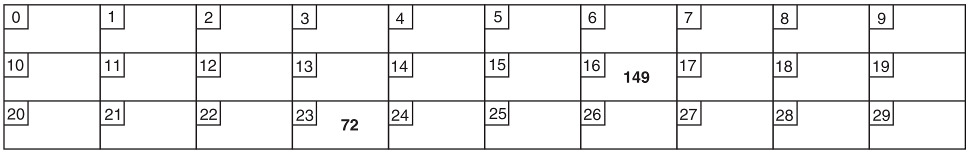
### Main Memory

Commonly known as *random access memory*, or *RAM*, the computer's main memory is a device that holds information. Specifically, RAM holds the sequences of instructions in the programs that are running and the data those programs are using.

Memory is divided into sections that hold an equal amount of data. Each section is made of eight "switches" that may be either on or off. A switch in the on position usually represents the number 1, whereas a switch in the off position usually represents the number 0. The computer stores data by setting the switches in a memory location to a pattern that represents a character or a number. Each of these switches is known as a *bit*, which stands for *binary digit*. Each section of memory, which is a collection of eight bits, is known as a *byte*. Each byte is assigned a unique number known as an *address*. The addresses are ordered from lowest to highest. A byte is identified by its address in much the same way a post office box is identified by an address. Figure 1-3 shows a series of bytes with their addresses. In the illustration, sample data is stored in memory. The number 149 is stored in the byte at address 16, and the number 72 is stored in the byte at address 23.

RAM is usually a volatile type of memory, used only for temporary storage. When the computer is turned off, the contents of RAM are erased.

**Figure 1-3** Memory bytes and their addresses



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 149 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 72 | 24 | 25 | 26 | 27 | 28 | 29 |

### Secondary Storage

Secondary storage is a type of memory that can hold data for long periods of time—even when there is no power to the computer. Frequently used programs are stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory figures, is saved to secondary storage as well.

The most common type of secondary storage device is the *disk drive*. A traditional disk drive stores data by magnetically encoding it onto a spinning circular disk. *Solid state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid-state drive has no moving parts, and operates faster than a traditional disk drive. Most computers have some sort of secondary storage device, either a traditional disk drive or a solid-state drive, mounted inside their case. External drives are also available, which connect to one of the computer's communication ports. External drives can be used to create backup copies of important data or to move data to another computer.

In addition to external drives, many types of devices have been created for copying data, and for moving it to other computers. *Universal Serial Bus drives,* or *USB drives* are small devices that plug into the computer's USB (Universal Serial Bus) port, and appear to the system as a disk drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also popular for data storage. Data is not recorded magnetically on an optical disc, but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they make a good medium for creating backup copies of data.

### Input Devices

Input is any data the computer collects from the outside world. The device that collects the data and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, scanner, and digital camera. Disk drives, optical drives, and USB drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

### Output Devices

Output is any data the computer sends to the outside world. It might be a sales report, a list of names, or a graphic image. The data is sent to an output device, which formats and presents it. Common output devices are monitors and printers. Disk drives, USB drives, and CD recorders can also be considered output devices because the CPU sends data to them to be saved.

## Software

As previously mentioned, software refers to the programs that run on a computer. There are two general categories of software: operating systems and application software. An operating system is a set of programs that manages the computer's hardware devices and controls their processes. Most all modern operating systems are multitasking, which means they are capable of running multiple programs at once. Through a technique called time sharing, a multitasking system divides the allocation of hardware resources and the attention of the CPU among all the executing programs. UNIX, Linux, Mac OS, and Windows are multitasking operating systems.

Application software refers to programs that make the computer useful to the user. These programs solve specific problems or perform general operations that satisfy the needs of the user. Word processing, spreadsheet, and database packages are all examples of application software.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

1.1 Why is the computer used by so many different people, in so many different professions?

1.2 List the five major hardware components of a computer system.

1.3 Internally, the CPU consists of what two units?

1.4 Describe the steps in the fetch/decode/execute cycle.

1.5 What is a memory address? What is its purpose?

1.6 Explain why computers have both main memory and secondary storage.

1.7 What does the term *multitasking* mean?

## 1.4 Programming Languages

**CONCEPT: A program is a set of instructions a computer follows in order to perform a task. A programming language is a special language used to write computer programs.**

### What Is a Program?

Computers are designed to follow instructions. A computer program is a set of instructions that enable the computer to solve a problem or perform a task. For example, suppose we want the computer to calculate someone's gross pay. The following is a list of things the computer should do to perform this task.

1. Display a message on the screen: "How many hours did you work?"
2. Allow the user to enter the number of hours worked.
3. Once the user enters a number, store it in memory.
4. Display a message on the screen: "How much do you get paid per hour?"
5. Allow the user to enter an hourly pay rate.
6. Once the user enters a number, store it in memory.
7. Once both the number of hours worked and the hourly pay rate are entered, multiply the two numbers and store the result in memory.
8. Display a message on the screen that shows the amount of money earned. The message must include the result of the calculation performed in Step 7.

Collectively, these instructions are called an *algorithm*. An algorithm is a set of well-defined steps for performing a task or solving a problem. Notice that these steps are sequentially ordered. Step 1 should be performed before Step 2, and so forth. It is important that these instructions be performed in their proper sequence.

Although you and I might easily understand the instructions in the pay-calculating algorithm, it is not ready to be executed on a computer. A computer's CPU can only process instructions that are written in *machine language*. If you were to look at a machine language program, you would see a stream of binary numbers (numbers consisting of only 1s and 0s). The binary numbers form machine language instructions, which the CPU interprets as commands. Here is an example of what a machine language instruction might look like:

```
1011010000000101
```

As you can imagine, the process of encoding an algorithm in machine language is very tedious and difficult. In addition, each different type of CPU has its own machine language. If you wrote a machine language program for computer A and then wanted to run it on computer B, which has a different type of CPU, you would have to rewrite the program in computer B's machine language.

Programming languages, which use words instead of numbers, were invented to ease the task of programming. A program can be written in a programming language, which is much easier to understand than machine language, and then translated into machine language. Programmers use software to perform this translation. Many programming languages have been created. Table 1-1 lists a few of the well-known ones.

**Table 1-1** Programming languages

| Language | Description |
|---|---|
| BASIC | Beginners All-purpose Symbolic Instruction Code is a general-purpose, procedural programming language. It was originally designed to be simple enough for beginners to learn. |
| FORTRAN | FORmula TRANslator is a procedural language designed for programming complex mathematical algorithms. |
| COBOL | Common Business-Oriented Language is a procedural language designed for business applications. |
| Pascal | Pascal is a structured, general-purpose, procedural language designed primarily for teaching programming. |
| C | C is a structured, general-purpose, procedural language developed at Bell Laboratories. |
| C++ | Based on the C language, C++ offers object-oriented features not found in C. C++ was also invented at Bell Laboratories. |
| C# | Pronounced "C sharp." It is a language invented by Microsoft for developing applications based on the Microsoft .NET platform. |
| Java | Java is an object-oriented language invented at Sun Microsystems, and is now owned by Oracle. It may be used to develop stand-alone applications that operate on a single computer, or applications that run over the Internet from a Web server. |
| JavaScript | JavaScript is a programming language that can be used in a Web site to perform simple operations. Despite its name, JavaScript is not related to Java. |
| Perl | A general-purpose programming language used widely on Internet servers. |
| PHP | A programming language used primarily for developing Web server applications and dynamic Web pages. |
| Python | Python is an object-oriented programming language used in both business and academia. Many popular Web sites contain features developed in Python. |
| Ruby | Ruby is a simple but powerful object-oriented programming language. It can be used for a variety of purposes, from small utility programs to large Web applications. |
| Visual Basic | Visual Basic is a Microsoft programming language and software development environment that allows programmers to create Windows-based applications quickly. |

## A History of Java

In 1991 a team was formed at Sun Microsystems (a company that is now owned by Oracle) to speculate about the important technological trends that might emerge in the near future. The team, which was named the Green Team, concluded that computers would merge with consumer appliances. Their first project was to develop a handheld device named *7 (pronounced star seven) that could be used to control a variety of home entertainment devices. For the unit to work, it had to use a programming language that could be processed by all the devices it controlled. This presented a problem because different brands of consumer devices use different processors, each with its own machine language.

Because no such universal language existed, James Gosling, the team's lead engineer, created one. Programs written in this language, which was originally named Oak, were not translated into the machine language of a specific processor, but were translated into an intermediate language known as *byte code*. Another program would then translate the byte code into machine language that could be executed by the processor in a specific consumer device.

Unfortunately, the technology developed by the Green Team was ahead of its time. No customers could be found, mostly because the computer-controlled consumer appliance industry was just beginning. But rather than abandoning their hard work and moving on to other projects, the team saw another opportunity: the Internet. The Internet is a perfect environment for a universal programming language such as Oak. It consists of numerous different computer platforms connected together in a single network.

To demonstrate the effectiveness of its language, which was renamed Java, the team used it to develop a Web browser. The browser, named HotJava, was able to download and run small Java programs known as applets. This gave the browser the capability to display animation and interact with the user. HotJava was demonstrated at the 1995 SunWorld conference before a wowed audience. Later the announcement was made that Netscape would incorporate Java technology into its Navigator browser. Other Internet companies rapidly followed, increasing the acceptance and the influence of the Java language. Today, Java is very popular for developing not only applets for developing Web applications, mobile apps, and desktop applications.

## 1.5 What Is a Program Made Of?

**CONCEPT: There are certain elements that are common to all programming languages.**

### Language Elements

All programming languages have some things in common. Table 1-2 lists the common elements you will find in almost every language.

**Table 1-2** The common elements of a programming language

| Language Element | Description |
|---|---|
| Key Words | These are words that have a special meaning in the programming language. They may be used for their intended purpose only. Key words are also known as reserved words. |
| Operators | Operators are symbols or words that perform operations on one or more operands. An operand is usually an item of data, such as a number. |
| Punctuation | Most programming languages require the use of punctuation characters. These characters serve specific purposes, such as marking the beginning or ending of a statement, or separating items in a list. |
| Programmer-Defined Names | Unlike key words, which are part of the programming language, these are words or names that are defined by the programmer. They are used to identify storage locations in memory and parts of the program that are created by the programmer. Programmer-defined names are often called identifiers. |
| Syntax | These are rules that must be followed when writing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear. |

Let's look at an example Java program and identify an instance of each of these elements. Code Listing 1-1 shows the code listing with each line numbered.

**NOTE:** The line numbers are not part of the program. They are included to help point out specific parts of the program.

**Code Listing 1-1** `Payroll.java`

```
 1  public class Payroll
 2  {
 3     public static void main(String[] args)
 4     {
 5        int hours = 40;
 6        double grossPay, payRate = 25.0;
 7
 8        grossPay = hours * payRate;
 9        System.out.println("Your gross pay is $" + grossPay);
10     }
11  }
```

### Key Words (Reserved Words)

Two of Java's key words appear in line 1: `public` and `class`. In line 3, the words `public`, `static`, and `void` are all key words. The words `int` in line 5 and `double` in line 6 are also key words. These words, which are always written in lowercase, each have a special meaning in Java and can only be used for their intended purpose. As you will see, the programmer is allowed to make up his or her own names for certain things in a program. Key words, however, are reserved and cannot be used for anything other than their designated purpose. Part of learning a programming language is learning the commonly used key words, what they mean, and how to use them.

Table 1-3 shows a list of the Java key words[1].

**Table 1-3** The Java key words

| | | | | | |
|---|---|---|---|---|---|
| `abstract` | `const` | `final` | `int` | `public` | `throw` |
| `assert` | `continue` | `finally` | `interface` | `return` | `throws` |
| `boolean` | `default` | `float` | `long` | `short` | `transient` |
| `break` | `do` | `for` | `native` | `static` | `true` |
| `byte` | `double` | `goto` | `new` | `strictfp` | `try` |
| `case` | `else` | `if` | `null` | `super` | `void` |
| `catch` | `enum` | `implements` | `package` | `switch` | `volatile` |
| `char` | `extends` | `import` | `private` | `synchronized` | `while` |
| `class` | `false` | `instanceof` | `protected` | `this` | |

### Programmer-Defined Names

The words `hours`, `payRate`, and `grossPay` that appear in the program in lines 5, 6, 8, and 9 are programmer-defined names. They are not part of the Java language but are names made up by the programmer. In this particular program, these are the names of variables. As you will learn later in this chapter, variables are the names of memory locations that may hold data.

### Operators

In line 8 the following line appears:

```
grossPay = hours * payRate;
```

The `=` and `*` symbols are both operators. They perform operations on items of data, known as operands. The `*` operator multiplies its two operands, which in this example are the variables `hours` and `payRate`. The `=` symbol is called the assignment operator. It takes the value of the expression that appears at its right and stores it in the variable whose name appears at its left. In this example, the `=` operator stores in the `grossPay` variable the result of the `hours` variable multiplied by the `payRate` variable. In other words, the statement says, "the `grossPay` variable is assigned the value of `hours` times `payRate`."

---

[1] Java 9 also introduces a set of *restricted words* that are treated as key words under certain circumstances. See Appendix C for the full list.

### Punctuation

Notice that lines 5, 6, 8, and 9 end with a semicolon. A semicolon in Java is similar to a period in English: It marks the end of a complete sentence (or statement, as it is called in programming jargon). Semicolons do not appear at the end of every line in a Java program, however. There are rules that govern where semicolons are required and where they are not. Part of learning Java is learning where to place semicolons and other punctuation symbols.

## Lines and Statements

Often, the contents of a program are thought of in terms of lines and statements. A *line* is just that—a single line as it appears in the body of a program. Code Listing 1-1 is shown with each of its lines numbered. Most of the lines contain something meaningful; however, line 7 is empty. Blank lines are only used to make a program more readable.

A statement is a complete instruction that causes the computer to perform some action. Here is the statement that appears in line 9 of Code Listing 1-1:

```
System.out.println("Your gross pay is $" + grossPay);
```

This statement causes the computer to display a message on the screen. Statements can be a combination of key words, operators, and programmer-defined names. Statements often occupy only one line in a program, but sometimes they are spread out over more than one line.

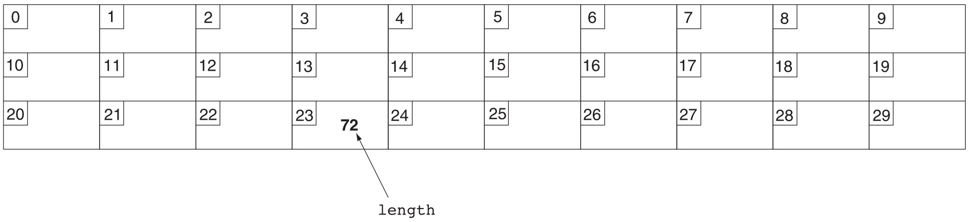
## Variables

The most fundamental way that a Java program stores an item of data in memory is with a variable. A *variable* is a named storage location in the computer's memory. The data stored in a variable may change while the program is running (hence the name "variable"). Notice that in Code Listing 1-1 the programmer-defined names `hours`, `payRate`, and `grossPay` appear in several places. All three of these are the names of variables. The `hours` variable is used to store the number of hours the user has worked. The `payRate` variable stores the user's hourly pay rate. The `grossPay` variable holds the result of `hours` multiplied by `payRate`, which is the user's gross pay.

Variables are symbolic names made up by the programmer that represent locations in the computer's RAM. When data is stored in a variable, it is actually stored in RAM. Assume that a program has a variable named `length`. Figure 1-4 illustrates the way the variable name represents a memory location.

In Figure 1-4, the variable `length` is holding the value 72. The number 72 is actually stored in RAM at address 23, but the name `length` symbolically represents this storage location. If it helps, you can think of a variable as a box that holds data. In Figure 1-4, the number 72 is stored in the box named `length`. Only one item may be stored in the box at any given time. If the program stores another value in the box, it will take the place of the number 72.

**Figure 1-4** A variable name represents a location in memory



## The Compiler and the Java Virtual Machine

When a Java program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The Java programming statements written by the programmer are called *source code*, and the file they are saved in is called a *source file*. Java source files end with the *.java* extension.

After the programmer saves the source code to a file, he or she runs the Java compiler. A *compiler* is a program that translates source code into an executable form. During the translation process, the compiler uncovers any syntax errors that may be in the program. *Syntax errors* are mistakes that the programmer has made that violate the rules of the programming language. These errors must be corrected before the compiler can translate the source code. Once the program is free of syntax errors, the compiler creates another file that holds the translated instructions.

Most programming language compilers translate source code directly into files that contain machine language instructions. These are called *executable files* because they may be executed directly by the computer's CPU. The Java compiler, however, translates a Java source file into a file that contains byte code instructions. Byte code instructions are not machine language, and therefore cannot be directly executed by the CPU. Instead, they are executed by the *Java Virtual Machine* (JVM). The JVM is a program that reads Java byte code instructions and executes them as they are read. For this reason, the JVM is often called an interpreter, and Java is often referred to as an interpreted language. Figure 1-5 illustrates the process of writing a Java program, compiling it to byte code, and running it.

Although Java byte code is not machine language for a CPU, it can be considered as machine language for the JVM. You can think of the JVM as a program that simulates a computer whose machine language is Java byte code.

### Portability

The term *portable* means that a program may be written on one type of computer and then run on a wide variety of computers, with little or no modification necessary. Because Java byte code is the same on all computers, compiled Java programs are highly portable. In fact, a compiled Java program may be run on any computer that has a Java Virtual Machine. Figure 1-6 illustrates the concept of a compiled Java program running on Windows, Linux, Mac, and UNIX computers.

With most other programming languages, portability is achieved by the creation of a compiler for each type of computer that the language is to run on. For example, in order for the C++ language to be supported by Windows, Linux, and Mac computers, a separate C++ compiler must be created for each of those environments. Compilers are very complex programs, and more difficult to develop than interpreters. For this reason, a JVM has been developed for many types of computers.

**Figure 1-5** Program development process

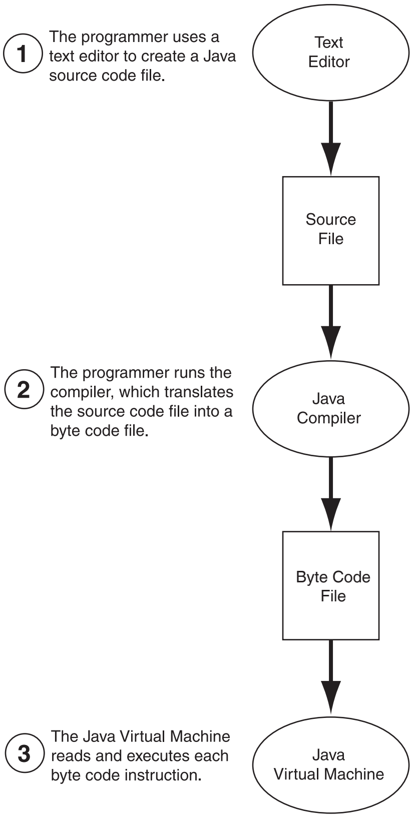
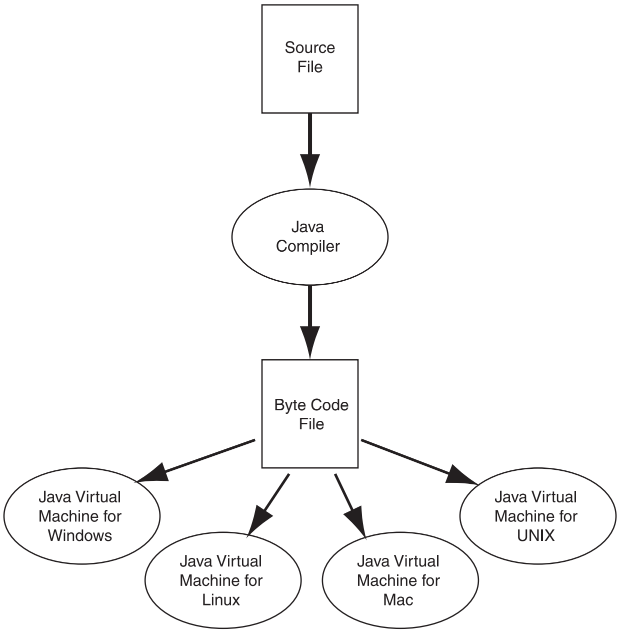

① The programmer uses a text editor to create a Java source code file.
Text Editor
Source File
② The programmer runs the compiler, which translates the source code file into a byte code file.
Java Compiler
Byte Code File
③ The Java Virtual Machine reads and executes each byte code instruction.
Java Virtual Machine

**Figure 1-6** Java byte code may be run on any computer with a Java Virtual Machine


Source File
Java Compiler
Byte Code File
Java Virtual Machine for Windows
Java Virtual Machine for Linux
Java Virtual Machine for Mac
Java Virtual Machine for UNIX

## Java Software Editions

The software that you use to create Java programs is referred to as the *JDK* (Java Development Kit) or the *SDK* (Software Development Kit). There are the following different editions of the JDK available from Oracle:

- *Java SE*—The Java Standard Edition provides all the essential software tools necessary for writing Java applications.
- *Java EE*—The Java Enterprise Edition provides tools for creating large business applications that employ servers and provide services over the Web.
- *Java ME*—The Java Micro Edition provides a small, highly optimized runtime environment for consumer products such as cell phones, pagers, and appliances.

These editions of Java may be downloaded from Oracle by going to:

http://java.oracle.com

**NOTE:** You can follow the instructions in Appendix D, which can be downloaded from the book's companion Web site, to install the JDK on your system. You can access the book's companion Web site by going to www.pearsonhighered.com/gaddis.

## Compiling and Running a Java Program

Compiling a Java program is a simple process. Once you have installed the JDK, go to your operating system's command prompt.

**TIP:** In Windows click Start, go to All Programs, and then go to Accessories. Click Command Prompt on the Accessories menu. A command prompt window should open.

VideoNote
Compiling and Running a Java Program

At the operating system command prompt, make sure you are in the same directory or folder where the Java program that you want to compile is located. Then, use the `javac` command, in the following form:

```
javac  Filename
```

`Filename` is the name of a file that contains the Java source code. As mentioned earlier, this file has the *.java* extension. For example, if you want to compile the *Payroll.java* file, you would execute the following command:

```
javac Payroll.java
```

This command runs the compiler. If the file contains any syntax errors, you will see one or more error messages and the compiler will not translate the file to byte code. When this happens you must open the source file in a text editor and fix the error. Then you can run the compiler again. If the file has no syntax errors, the compiler will translate it to byte code. Byte code is stored in a file with the *.class* extension, so the byte code for the *Payroll.java* file will be stored in *Payroll.class*, which will be in the same directory or folder as the source file.

To run the Java program, you use the `java` command in the following form:

```
java  ClassFilename
```

`ClassFilename` is the name of the *.class* file that you wish to execute; however, you do not type the *.class* extension. For example, to run the program that is stored in the *Payroll.class* file, you would enter the following command:

```
java Payroll
```

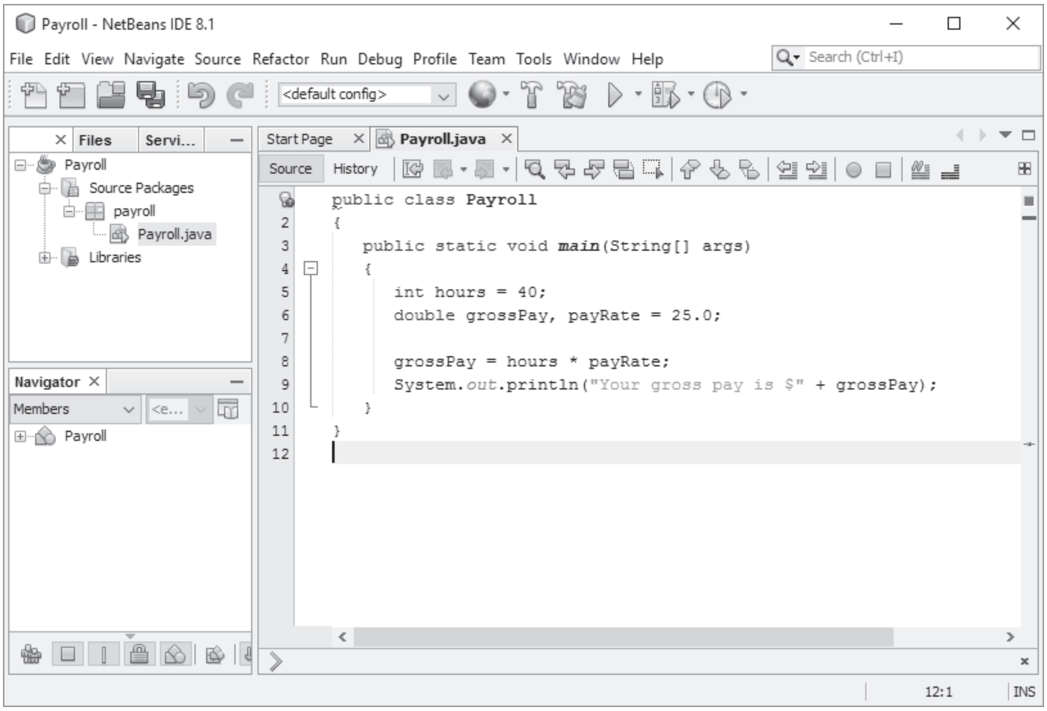
This command runs the Java interpreter (the JVM) and executes the program.

VideoNote
Using an IDE

### Integrated Development Environments

In addition to the command prompt programs, there are also several Java integrated development environments (IDEs). These environments consist of a text editor, compiler, debugger, and other utilities integrated into a package with a single set of menus. A program is compiled and executed with a single click of a button, or by selecting a single item from a menu. Figure 1-7 shows a screen from the NetBeans IDE.

**Figure 1-7** An integrated development environment (IDE) (Oracle Corporate Counsel)


Payroll - NetBeans IDE 8.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Files
Servi...
Payroll
Source Packages
payroll
Payroll.java
Libraries
Navigator
Members
<e...
Payroll
Start Page
Payroll.java
Source
History
12:1
INS

```
public class Payroll
{
   public static void main(String[] args)
   {
      int hours = 40;
      double grossPay, payRate = 25.0;

      grossPay = hours * payRate;
      System.out.println("Your gross pay is $" + grossPay);
   }
}
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

1.8 Describe the difference between a key word and a programmer-defined symbol.

1.9 Describe the difference between operators and punctuation symbols.

1.10 Describe the difference between a program line and a statement.

1.11 Why are variables called “variable”?

1.12 What happens to a variable’s current contents when a new value is stored there?

1.13 What is a compiler?

1.14 What is a syntax error?

1.15 What is byte code?

1.16 What is the JVM?

## 1.6 The Programming Process

**CONCEPT: The programming process consists of several steps, which include design, creation, testing, and debugging activities.**

Now that you have been introduced to what a program is, it's time to consider the process of creating a program. Quite often when inexperienced students are given programming assignments, they have trouble getting started because they don't know what to do first. If you find yourself in this dilemma, the following steps may help.

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.
4. Check the model for logical errors.
5. Enter the code and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any runtime errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

These steps emphasize the importance of planning. Just as there are good ways and bad ways to paint a house, there are good ways and bad ways to create a program. A good program always begins with planning. With the pay-calculating algorithm that was presented earlier in this chapter serving as our example, let's look at each of the steps in more detail.

### 1. Clearly define what the program is to do

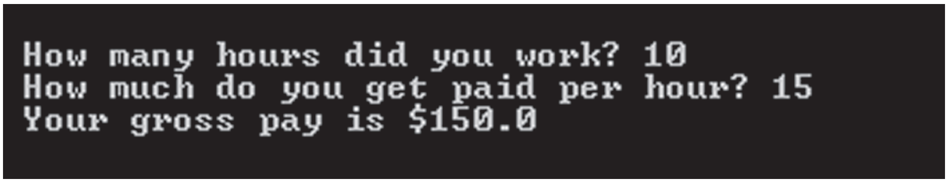
This step commonly requires you to identify the purpose of the program, the data that is to be input, the processing that is to take place, and the desired output. Let's examine each of these requirements for the pay-calculating algorithm.

| | |
|---|---|
| *Purpose* | To calculate the user's gross pay. |
| *Input* | Number of hours worked, hourly pay rate. |
| *Process* | Multiply number of hours worked by hourly pay rate. The result is the user's gross pay. |
| *Output* | Display a message indicating the user's gross pay. |

### 2. Visualize the program running on the computer

Before you create a program on the computer, you should first create it in your mind. Try to imagine what the computer screen will look like while the program is running. If it helps, draw pictures of the screen, with sample input and output, at various points in the program. For instance, Figure 1-8 shows the screen we might want produced by a program that implements the pay-calculating algorithm.

**Figure 1-8** Screen produced by the pay-calculating algorithm



```
How many hours did you work? 10
How much do you get paid per hour? 15
Your gross pay is $150.0
```

In this step, you must put yourself in the shoes of the user. What messages should the program display? What questions should it ask? By addressing these concerns, you can determine most of the program's output.

### 3. Use design tools to create a model of the program

While planning a program, the programmer uses one or more design tools to create a model of the program. For example, *pseudocode* is a cross between human language and a programming language and is especially helpful when designing an algorithm. Although the computer can't understand pseudocode, programmers often find it helpful to write an algorithm in a language that's "almost" a programming language, but still very similar to natural language. For example, here is pseudocode that describes the pay-calculating algorithm:

```
Get payroll data.
Calculate gross pay.
Display gross pay.
```

Although this pseudocode gives a broad view of the program, it doesn't reveal all the program's details. A more detailed version of the pseudocode follows:

```
Display "How many hours did you work?"
Input hours.
Display "How much do you get paid per hour?"
Input rate.
Store the value of hours times rate in the pay variable.
Display the value in the pay variable.
```

Notice that the pseudocode uses statements that look more like commands than the English statements that describe the algorithm in Section 1.4. The pseudocode even names variables and describes mathematical operations.

### 4. Check the model for logical errors

Logical errors are mistakes that cause the program to produce erroneous results. Once a model of the program is assembled, it should be checked for these errors. For example, if pseudocode is used, the programmer should trace through it, checking the logic of each step. If an error is found, the model can be corrected before the next step is attempted.

### 5. Enter the code and compile it

Once a model of the program has been created, checked, and corrected, the programmer is ready to write source code on the computer. The programmer saves the source code to a file and begins the process of compiling it. During this step the compiler will find any syntax errors that may exist in the program.

### 6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary

If the compiler reports any errors, they must be corrected. Steps 5 and 6 must be repeated until the program is free of compile-time errors.

### 7. Run the program with test data for input

Once an executable file is generated, the program is ready to be tested for runtime errors. A runtime error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

Testing for runtime errors requires that the program be executed with sample data or sample input. The sample data should be such that the correct output can be predicted. If the program does not produce the correct output, a logical error is present in the program.

### 8. Correct any runtime errors found while running the program. Repeat Steps 5 through 8 as many times as necessary

When runtime errors are found in a program, they must be corrected. You must identify the step where the error occurred and determine the cause. If an error is a result of incorrect logic (such as an improperly stated math formula), you must correct the statement or statements involved in the logic. If an error is due to an incomplete understanding of the program requirements, then you must restate the program purpose and modify the program model and source code. The program must then be saved, recompiled, and retested. This means Steps 5 though 8 must be repeated until the program reliably produces satisfactory results.

### 9. Validate the results of the program

When you believe you have corrected all the runtime errors, enter test data and determine whether the program solves the original problem.

## Software Engineering

The field of software engineering encompasses the whole process of crafting computer software. It includes designing, writing, testing, debugging, documenting, modifying, and maintaining complex software development projects. Like traditional engineers, software engineers use a number of tools in their craft. Here are a few examples:

- Program specifications
- Diagrams of screen output
- Diagrams representing the program components and the flow of data
- Pseudocode
- Examples of expected input and desired output
- Special software designed for testing programs

Most commercial software applications are large and complex. Usually a team of programmers, not a single individual, develops them. It is important that the program requirements be thoroughly analyzed and divided into subtasks that are handled by individual teams, or individuals within a team.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

1.17 What four items should you identify when defining what a program is to do?

1.18 What does it mean to "visualize a program running"? What is the value of such an activity?

1.19 What is pseudocode?

1.20 Describe what a compiler does with a program's source code.

1.21 What is a runtime error?

1.22 Is a syntax error (such as misspelling a key word) found by the compiler or when the program is running?

1.23 What is the purpose of testing a program with sample data or input?

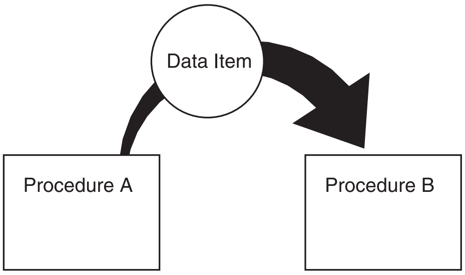
## 1.7 Object-Oriented Programming

**CONCEPT: Java is an object-oriented programming (OOP) language. OOP is a method of software development that has its own practices, concepts, and vocabulary.**

There are primarily two methods of programming in use today: procedural and object-oriented. The earliest programming languages were procedural, meaning a program was made of one or more procedures. A *procedure* is a set of programming statements that, together, perform a specific task. The statements might gather input from the user, manipulate data stored in the computer's memory, and perform calculations or any other operation necessary to complete the procedure's task.

Procedures typically operate on data items that are separate from the procedures. In a procedural program, the data items are commonly passed from one procedure to another, as shown in Figure 1-9.

**Figure 1-9** Data is passed among procedures



As you might imagine, the focus of procedural programming is on the creation of procedures that operate on the program's data. The separation of data and the code that operates on the data often leads to problems, however. For example, the data is stored in a particular format, which consists of variables and more complex structures that are created from variables. The procedures that operate on the data must be designed with that format in mind. But, what happens if the format of the data is altered? Quite often, a program's specifications change, resulting in a redesigned data format. When the structure of the data changes, the code that operates on the data must also be changed to accept the new format. This results in added work for programmers and a greater opportunity for bugs to appear in the code.

This has helped influence the shift from procedural programming to object-oriented programming (OOP). Whereas procedural programming is centered on creating procedures, object-oriented programming is centered on creating objects. An object is a software entity that contains data and procedures. The data contained in an object is known as the object's *attributes*. The procedures, or behaviors, that an object performs are known as the object's *methods*. The object is, conceptually, a self-contained unit consisting of data (attributes) and procedures (methods). This is illustrated in Figure 1-10.

OOP addresses the problem of code/data separation through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data from code that is outside the object. Only the object's methods may then directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access the methods that operate on the data. As shown in Figure 1-11, the object's methods provide programming statements outside the object with indirect access to the object's data.

When an object's internal data is hidden from outside code and access to that data is restricted to the object's methods, the data is protected from accidental corruption. In addition, the programming code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's methods.

**Figure 1-10** An object contains data and procedures

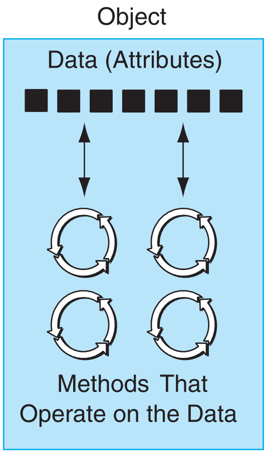
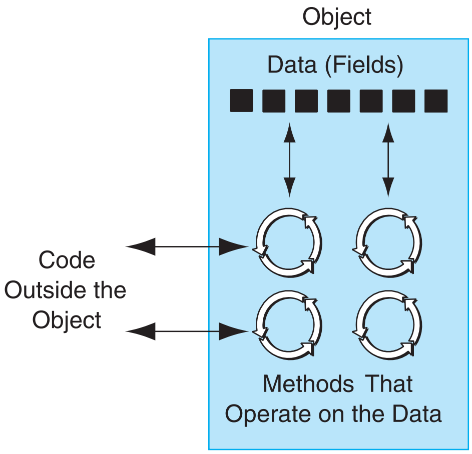


**Figure 1-11** Code outside the object interacts with the object's methods



When a programmer changes the structure of an object's internal data, he or she also modifies the object's methods so they may properly operate on the data. The way in which outside code interacts with the methods, however, does not change.

These are just a few of the benefits of object-oriented programming. Because Java is fully object-oriented, you will learn much more about OOP practices, concepts, and terms as you progress through this book.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

1.24 In procedural programming, what two parts of a program are typically separated?

1.25 What are an object's attributes?

1.26 What are an object's methods?

1.27 What is encapsulation?

1.28 What is data hiding?

## Review Questions and Exercises

### Multiple Choice

1. This part of the computer fetches instructions, carries out the operations commanded by the instructions, and produces some outcome or resultant information.
   a. memory
   b. CPU
   c. secondary storage
   d. input device
2. A byte is made up of eight
   a. CPUs
   b. addresses
   c. variables
   d. bits
3. Each byte is assigned a unique
   a. address
   b. CPU
   c. bit
   d. variable
4. This type of memory can hold data for long periods of time—even when there is no power to the computer.
   a. RAM
   b. primary storage
   c. secondary storage
   d. CPU storage

5. If you were to look at a machine language program, you would see _______.
   a. Java source code
   b. a stream of binary numbers
   c. English words
   d. circuits
6. These are words that have a special meaning in the programming language.
   a. punctuation
   b. programmer-defined names
   c. key words
   d. operators
7. These are symbols or words that perform operations on one or more operands.
   a. punctuation
   b. programmer-defined names
   c. key words
   d. operators
8. These characters serve specific purposes, such as marking the beginning or ending of a statement, or separating items in a list.
   a. punctuation
   b. programmer-defined names
   c. key words
   d. operators
9. These are words or names that are used to identify storage locations in memory and parts of the program that are created by the programmer.
   a. punctuation
   b. programmer-defined names
   c. key words
   d. operators
10. These are the rules that must be followed when writing a program.
    a. syntax
    b. punctuation
    c. key words
    d. operators
11. This is a named storage location in the computer's memory.
    a. class
    b. key word
    c. variable
    d. operator
12. The Java compiler generates _______.
    a. machine code
    b. byte code
    c. source code
    d. HTML
13. JVM stands for _______.
    a. Java Variable Machine
    b. Java Variable Method
    c. Java Virtual Method
    d. Java Virtual Machine

## Find the Error

1. The following pseudocode algorithm has an error. The program is supposed to ask the user for the length and width of a rectangular room, and then display the room's area. The program must multiply the width by the length to determine the area. Find the error.

```
area = width × length
Display "What is the room's width?"
Input width
Display "What is the room's length?"
Input length
Display area
```

## Algorithm Workbench

Write pseudocode algorithms for the programs described as follows:

1. **Available Credit**

   A program that calculates a customer's available credit should ask the user for the following:

   - The customer's maximum amount of credit
   - The amount of credit used by the customer

   Once these items have been entered, the program should calculate and display the customer's available credit. You can calculate available credit by subtracting the amount of credit used from the maximum amount of credit.

2. **Sales Tax**

   A program that calculates the total of a retail sale should ask the user for the following:

   - The retail price of the item being purchased
   - The sales tax rate

   Once these items have been entered, the program should calculate and display the following:

   - The sales tax for the purchase
   - The total of the sale

3. **Account Balance**

   A program that calculates the current balance in a savings account must ask the user for the following:

   The starting balance
   The total dollar amount of deposits made
   The total dollar amount of withdrawals made
   The monthly interest rate

   Once the program calculates the current balance, it should be displayed on the screen.

### Predict the Result

The following are programs expressed as English statements. What would each display on the screen if they were actual programs?

1. The variable `x` starts with the value 0.

   The variable `y` starts with the value 5.
   Add 1 to `x`.
   Add 1 to `y`.
   Add `x` and `y`, and store the result in `y`.
   Display the value in `y` on the screen.
2. The variable `a` starts with the value 10.

   The variable `b` starts with the value 2.
   The variable `c` starts with the value 4.
   Store the value of `a` times `b` in `a`.
   Store the value of `b` times `c` in `c`.
   Add `a` and `c`, and store the result in `b`.
   Display the value in `b` on the screen.

### Short Answer

1. Both main memory and secondary storage are types of memory. Describe the difference between the two.
2. What type of memory is usually volatile?
3. What is the difference between operating system software and application software?
4. Why must programs written in a high-level language be translated into machine language before they can be run?
5. Why is it easier to write a program in a high-level language than in machine language?
6. What is a source file?
7. What is the difference between a syntax error and a logical error?
8. What is an algorithm?
9. What is a compiler?
10. What must a computer have in order for it to execute Java programs?
11. What is the difference between machine language code and byte code?
12. Why does byte code make Java a portable language?
13. Is encapsulation a characteristic of procedural or object-oriented programming?
14. Why should an object hide its data?
15. What part of an object forms an interface through which outside code may access the object's data?
16. What type of program do you use to write Java source code?
17. Will the Java compiler translate a source file that contains syntax errors?
18. What does the Java compiler translate Java source code to?
19. Assuming you are using the JDK, what command would you type at the operating system command prompt to compile the program `LabAssignment.java`?

20. Assuming there are no syntax errors in the *LabAssignment.java* program when it is compiled, answer the following questions.
    - a. What file will be produced?
    - b. What will the file contain?
    - c. What command would you type at the operating system command prompt to run the program?

## Programming Challenge

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Your First Java Program

This assignment will help you get acquainted with your Java development software. Here is the Java program you will enter:

VideoNote
Your First Java Program

```
// This is my first Java program.
public class MyFirstProgram
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

### If You Are Using the JDK at the Command Prompt:

1. Use a text editor to type the source code exactly as it is shown. Be sure to place all the punctuation characters and be careful to match the case of the letters as they are shown. Save it to a file named *MyFirstProgram.java*.
2. After saving the program, go to your operating system's command prompt and change your current directory or folder to the one that contains the Java program you just created. Then use the following command to compile the program:

   ```
   javac MyFirstProgram.java
   ```

   If you typed the contents of the file exactly as shown, you shouldn't have any syntax errors. If you see error messages, open the file in the editor and compare your code to that shown. Correct any mistakes you have made, save the file, and run the compiler again. If you see no error messages, the file was successfully compiled.
3. Next, enter the following command to run the program:

   ```
   java MyFirstProgram
   ```

   Be sure to use the capitalization of `MyFirstProgram` exactly as it is shown here. You should see the message "Hello World!" displayed on the screen.

**If You Are Using an IDE:**

*Because there are many Java IDEs, we cannot include specific instructions for all of these. The following are general steps that should apply to most of them. You will need to consult your IDE's documentation for specific instructions.*

1. Start your Java IDE and perform any necessary setup operations, such as starting a new project and creating a new Java source file.
2. Use the IDE's text editor to type the source code exactly as it is shown. Be sure to place all the punctuation characters and be careful to match the case of the letters as they are shown. Save it to a file named *MyFirstProgram.java*.
3. After saving the program, use your IDE's command to compile the program. If you typed the contents of the file exactly as shown, you shouldn't have any syntax errors. If you see error messages, compare your code to that shown. Correct any mistakes you have made, save the file, and run the compiler again. If you see no error messages, the file was successfully compiled.

   Use your IDE's command to run the program. You should see the message "Hello World!" displayed.

CHAPTER

# 2 Java Fundamentals

## TOPICS

2.1 The Parts of a Java Program
2.2 The `print` and `println` Methods, and the Java API
2.3 Variables and Literals
2.4 Primitive Data Types
2.5 Arithmetic Operators
2.6 Combined Assignment Operators
2.7 Conversion between Primitive Data Types
2.8 Creating Named Constants with `final`
2.9 The `String` Class
2.10 Scope
2.11 Comments
2.12 Programming Style
2.13 Reading Keyboard Input
2.14 Dialog Boxes
2.15 Common Errors to Avoid

## 2.1 The Parts of a Java Program

**CONCEPT:** **A Java program has parts that serve specific purposes.**

Java programs are made up of different parts. Your first step in learning Java is to learn what the parts are. We will begin by looking at a simple example, shown in Code Listing 2-1.

**Code Listing 2-1** (`Simple.java`)

```
// This is a simple Java program.

public class Simple
{
   public static void main(String[] args)
   {
      System.out.println("Programming is great fun!");
   }
}
```

**TIP:** Remember, the line numbers shown in the program listings are not part of the program. The numbers are shown so we can refer to specific lines in the programs.

As mentioned in Chapter 1, the names of Java source code files end with *.java*. The program shown in Code Listing 2-1 is named *Simple.java*. Using the Java compiler, this program may be compiled with the following command:

```
javac Simple.java
```

The compiler will create another file named *Simple.class*, which contains the translated Java byte code. This file can be executed with the following command:

```
java Simple
```

**TIP:** Remember, you do not type the *.class* extension when using the `java` command.

The output of the program is as follows. This is what appears on the screen when the program runs.

**Program Output**

```
Programming is great fun!
```

Lets's examine the program line by line. Here's the statement in line 1:

```
// This is a simple Java program.
```

Other than the two slash marks that begin this line, it looks pretty much like an ordinary sentence. The `//` marks the beginning of a comment. The compiler ignores everything from the double-slash to the end of the line. That means you can type anything you want on that line and the compiler never complains. Although comments are not required, they are very important to programmers. Most programs are much more complicated than this example, and comments help explain what's going on.

Line 2 is blank. Programmers often insert blank lines in programs to make them easier to read. Line 3 reads:

```
public class Simple
```

This line is known as a *class header*, and it marks the beginning of a *class definition*. One of the uses of a class is to serve as a container for an application. As you progress through this book, you will learn more and more about classes. For now, just remember that a Java program must have at least one class definition. This line of code consists of three words: `public`, `class`, and `Simple`. Let's take a closer look at each word.

- `public` is a Java key word, and it must be written in all lowercase letters. It is known as an *access specifier*, and it controls where the class may be accessed from. The `public` specifier means access to the class is unrestricted. (In other words, the class is "open to the public.")
- `class`, which must also be written in lowercase letters, is a Java key word that indicates the beginning of a class definition.

- `Simple` is the class name. This name was made up by the programmer. The class could have been called `Pizza`, or `Dog`, or anything else the programmer wanted. Programmer-defined names may be written in lowercase letters, uppercase letters, or a mixture of both.

In a nutshell, this line of code tells the compiler that a publicly accessible class named `Simple` is being defined. Here are two more points to know about classes:

- You may create more than one class in a file, but you may have only one `public class` per Java file.
- When a Java file has a `public` class, the name of the public class must be the same as the name of the file (without the *.java* extension). For instance, the program in Code Listing 2-1 has a `public class` named `Simple`, so it is stored in a file named *Simple.java*.

**NOTE:** Java is a case-sensitive language. That means it regards uppercase letters as being entirely different characters than their lowercase counterparts. The word `Public` is not the same as `public`, and `Class` is not the same as `class`. Some words in a Java program must be entirely in lowercase, while other words may use a combination of lower and uppercase characters. Later in this chapter, you will see a list of all the Java key words, which must appear in lowercase.

Line 4 contains only a single character:

```
{
```

This is called a left brace, or an opening brace, and is associated with the beginning of the class definition. All of the programming statements that are part of the class are enclosed in a set of braces. If you glance at the last line in the program, line 9, you'll see the closing brace. Everything between the two braces is the *body* of the class named `Simple`. Here is the program code again, this time the body of the class definition is shaded.

```
// This is a simple Java program.
public class Simple
{
   public static void main(String[] args)
   {
      System.out.println("Programming is great fun!");
   }
}
```

**WARNING!** Make sure you have a closing brace for every opening brace in your program!

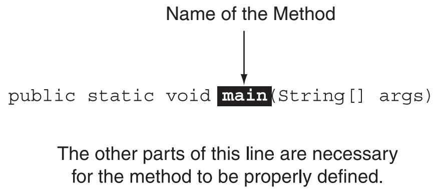
Line 5 reads:

```
public static void main(String[] args)
```

This line is known as a *method header*. It marks the beginning of a *method*. A method can be thought of as a group of one or more programming statements that collectively has a name. When creating a method, you must tell the compiler several things about it. That is why this line contains so many words. At this point, the only thing you should be concerned about is that the name of the method is `main`, and the rest of the words are required for the method to be properly defined. This is shown in Figure 2-1.

Recall from Chapter 1 that a stand-alone Java program that runs on your computer is known as an application. Every Java application must have a method named `main`. The `main` method is the starting point of an application.

**Figure 2-1** The `main` method header


Name of the Method
The other parts of this line are necessary for the method to be properly defined.

```
public static void main(String[] args)
```

**NOTE:** For the time being, all the programs you will write will consist of a class with a `main` method whose header looks exactly like the one shown in Code Listing 2-1. As you progress through this book you will learn what `public static void` and `(String[] args)` mean. For now, just assume that you are learning a "recipe" for assembling a Java program.

Line 6 has another opening brace:

```
{
```

This opening brace belongs to the `main` method. Remember that braces enclose statements, and every opening brace must have an accompanying closing brace. If you look at line 8 you will see the closing brace that corresponds with this opening brace. Everything between these braces is the *body* of the `main` method.

Line 7 appears as follows:

```
System.out.println("Programming is great fun!");
```

To put it simply, this line displays a message on the screen. The message, "Programming is great fun!" is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal*.

**NOTE:** This is the only line in the program that causes anything to be printed on the screen. The other lines, like `public class Simple` and `public static void main(String[] args)`, are necessary for the framework of your program, but they do not cause any screen output. Remember, a program is a set of instructions for the computer. If something is to be displayed on the screen, you must use a programming statement for that purpose.

At the end of the line is a *semicolon*. Just as a period marks the end of a sentence, a semicolon marks the end of a statement in Java. Not every line of code ends with a semicolon, however. Here is a summary of where you do not place a semicolon:

- Comments do not have to end with a semicolon because they are ignored by the compiler.
- Class headers and method headers do not end with a semicolon because they are terminated with a body of code inside braces.
- The brace characters, `{` and `}`, are not statements, so you do not place a semicolon after them.

It might seem that the rules for where to put a semicolon are not clear at all. For now, just concentrate on learning the parts of a program. You'll soon get a feel for where you should and should not use semicolons.

As has already been pointed out, lines 8 and 9 contain the closing braces for the `main` method and the class definition:

```
    }
}
```

Before continuing, let's review the points we just covered, including some of the more elusive rules.

- Java is a case-sensitive language. It does not regard uppercase letters as being the same character as their lowercase equivalents.
- All Java programs must be stored in a file with a name that ends with *.java*.
- Comments are ignored by the compiler.
- A *.java* file may contain many classes, but may have only one `public` class. If a *.java* file has a public class, the class must have the same name as the file. For instance, if the file *Pizza.java* contains a `public class`, the class's name would be `Pizza`.
- Every Java application program must have a method named `main`.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons. This does not include comments, class headers, method headers, or braces.

In the sample program, you encountered several special characters. Table 2-1 summarizes how they were used.

**Table 2-1** Special characters

| Characters | Name | Meaning |
|---|---|---|
| `//` | Double slash | Marks the beginning of a comment |
| `( )` | Opening and closing parentheses | Used in a method header |
| `{ }` | Opening and closing braces | Encloses a group of statements, such as the contents of a class or a method |
| `" "` | Quotation marks | Encloses a string of characters, such as a message that is to be printed on the screen |
| `;` | Semicolon | Marks the end of a complete programming statement |

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.1 The following program will not compile because the lines have been mixed up.

```
public static void main(String[] args)
}
// A crazy mixed up program
public class Columbus
{
System.out.println("In 1492 Columbus sailed the ocean blue.");
{
}
```

When the lines are properly arranged, the program should display the following on the screen:

```
In 1492 Columbus sailed the ocean blue.
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.2 When the program in Question 2.1 is saved to a file, what should the file be named?

2.3 Complete the following program skeleton so it displays the message "Hello World" on the screen.

```
public class Hello
{
    public static void main(String[] args)
    {
        // Insert code here to complete the program
    }
}
```

2.4 On paper, write a program that will display your name on the screen. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

2.5 All Java source code filenames must end with __________.

a) a semicolon
b) *.class*
c) *.java*
d) none of the above

2.6 Every Java application program must have __________.

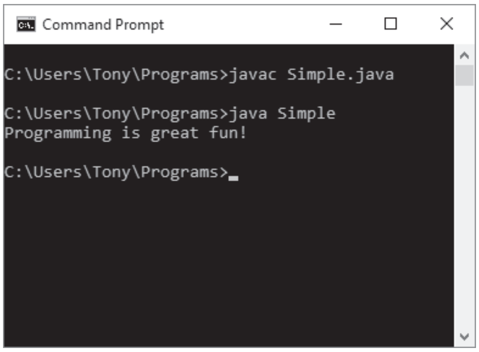
a) a method named `main`
b) more than one class definition
c) one or more comments

## 2.2 The `print` and `println` Methods, and the Java API

**CONCEPT: The `print` and `println` methods are used to display text output. They are part of the Java API, which is a collection of prewritten classes and methods for performing specific operations.**

In this section, you will learn how to write programs that produce output on the screen. The simplest type of output that a program can display on the screen is console output. *Console output* is merely plain text. When you display console output in a system that uses a graphical user interface, such as Windows or Mac OS, the output usually appears in a window similar to the one shown in Figure 2-2.

**Figure 2-2** A console window (Microsoft Corporation)



```
C:\Users\Tony\Programs>javac Simple.java

C:\Users\Tony\Programs>java Simple
Programming is great fun!

C:\Users\Tony\Programs>_
```

**VideoNote** Displaying Console Output

The word *console* is an old computer term. It comes from the days when the operator of a large computer system interacted with the system by typing on a terminal that consisted of a simple screen and keyboard. This terminal was known as the *console*. The console screen, which displayed only text, was known as the standard output device. Today, the term *standard output device* typically refers to the device that displays console output.

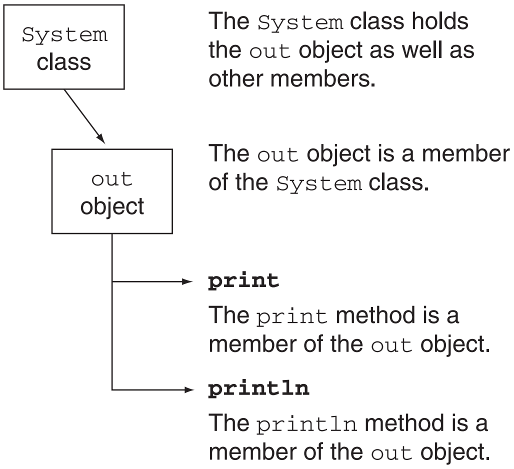
Performing output in Java, as well as many other tasks, is accomplished by using the Java API. The term *API* stands for *Application Programmer Interface*. The API is a standard library of prewritten classes for performing specific operations. These classes and their methods are available to all Java programs. The `print` and `println` methods are part of the API and provide ways for output to be displayed on the standard output device.

The program in Code Listing 2-1 (`Simple.java`) uses the following statement to print a message on the screen:

```
System.out.println("Programming is great fun!");
```

`System` is a class that is part of the Java API. The `System` class contains objects and methods that perform system-level operations. One of the objects contained in the `System` class is named `out`. The `out` object has methods, such as `print` and `println`, for performing output on the system console, or standard output device. The hierarchical relationship among `System`, `out`, `print`, and `println` is shown in Figure 2-3.

**Figure 2-3** Relationship among the `System` class, the `out` object, and the `print` and `println` methods


System class
The System class holds the out object as well as other members.
out object
The out object is a member of the System class.
print
The print method is a member of the out object.
println
The println method is a member of the out object.

Here is a brief summary of how it all works together:

- The `System` class is part of the Java API. It has member objects and methods for performing system-level operations, such as sending output to the console.
- The `out` object is a member of the `System` class. It provides methods for sending output to the screen.
- The `print` and `println` methods are members of the `out` object. They actually perform the work of writing characters on the screen.

This hierarchy explains why the statement that executes `println` is so long. The sequence `System.out.println` specifies that `println` is a member of `out`, which is a member of `System`.

**NOTE:** The period that separates the names of the objects is pronounced "dot." `System.out.println` is pronounced "system dot out dot print line."

The value that is to be displayed on the screen is placed inside the parentheses. This value is known as an *argument*. For example, the following statement executes the `println` method using the string `"King Arthur"` as its argument. This will print "King Arthur" on the screen. (The quotation marks are not displayed.)

```
System.out.println("King Arthur");
```

An important thing to know about the `println` method is that after it displays its message, it advances the cursor to the beginning of the next line. The next item printed on the screen will begin in this position. For example, look at the program in Code Listing 2-2.

Because each string is printed with separate `println` statements in Code Listing 2-2, they appear on separate lines in the Program Output.

**Code Listing 2-2** (`TwoLines.java`)

```
// This is another simple Java program.

public class TwoLines
{
   public static void main(String[] args)
   {
      System.out.println("Programming is great fun!");
      System.out.println("I can't get enough of it!");
   }
}
```

**Program Output**

```
Programming is great fun!
I can't get enough of it!
```

### The `print` Method

The `print` method, which is also part of the `System.out` object, serves a purpose similar to that of `println`—to display output on the screen. The `print` method, however, does not advance the cursor to the next line after its message is displayed. Look at Code Listing 2-3.

**Code Listing 2-3** (`GreatFun.java`)

```
// This is another simple Java program.

public class GreatFun
{
   public static void main(String[] args)
   {
      System.out.print("Programming is ");
      System.out.println("great fun!");
   }
}
```

**Program Output**

```
Programming is great fun!
```

An important concept to understand about Code Listing 2-3 is that, although the output is broken up into two programming statements, this program will still display the message on one line. The data that you send to the `print` method is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. The program in Code Listing 2-4 is an example.

**Code Listing 2-4** `(Unruly.java)`

```java
 1  // An unruly printing program
 2
 3  public class Unruly
 4  {
 5     public static void main(String[] args)
 6     {
 7        System.out.print("These are our top sellers:");
 8        System.out.print("Computer games");
 9        System.out.print("Coffee");
10        System.out.println("Aspirin");
11     }
12  }
```

**Program Output**

```
These are our top sellers:Computer gamesCoffeeAspirin
```

The layout of the actual output looks nothing like the arrangement of the strings in the source code. First, even though the output is broken up into four lines in the source code (lines 7 through 10), it comes out on the screen as one line. Second, notice that some of the words that are displayed are not separated by spaces. The strings are displayed exactly as they are sent to the `print` method. If spaces are to be displayed, they must appear in the strings.

There are two ways to fix this program. The most obvious way is to use `println` methods instead of `print` methods. Another way is to use escape sequences to separate the output into different lines. An *escape sequence* starts with the backslash character (`\`), and is followed by one or more *control characters*. It allows you to control the way output is displayed by embedding commands within the string itself. The escape sequence that causes the output cursor to go to the next line is `\n`. Code Listing 2-5 illustrates its use.

**Code Listing 2-5** `(Adjusted.java)`

```java
// A well adjusted printing program

public class Adjusted
{
   public static void main(String[] args)
   {
      System.out.print("These are our top sellers:\n");
      System.out.print("Computer games\nCoffee\n");
      System.out.println("Aspirin");
   }
}
```

**Program Output**

```
These are our top sellers:
Computer games
Coffee
Aspirin
```

The `\n` characters are called the newline escape sequence. When the `print` or `println` method encounters `\n` in a string, it does not print the `\n` characters on the screen, but interprets them as a special command to advance the output cursor to the next line. There are several other escape sequences as well. For instance, `\t` is the tab escape sequence. When `print` or `println` encounters it in a string, it causes the output cursor to advance to the next tab position. Code Listing 2-6 shows it in use.

**Code Listing 2-6** (`Tabs.java`)

```java
// Another well-adjusted printing program

public class Tabs
{
   public static void main(String[] args)
   {
      System.out.print("These are our top sellers:\n");
      System.out.print("\tComputer games\n\tCoffee\n");
      System.out.println("\tAspirin");
   }
}
```

**Program Output**

```
These are our top sellers:
        Computer games
        Coffee
        Aspirin
```

**NOTE:** Although you have to type two characters to write an escape sequence, they are stored in memory as a single character.

Table 2-2 lists the common escape sequences and describes them.

**Table 2-2** Common escape sequences

| Escape Sequence | Name | Description |
|---|---|---|
| `\n` | Newline | Advances the cursor to the next line for subsequent printing |
| `\t` | Horizontal tab | Causes the cursor to skip over to the next tab stop |
| `\b` | Backspace | Causes the cursor to back up, or move left, one position |
| `\r` | Return | Causes the cursor to go to the beginning of the current line, not the next line |
| `\\` | Backslash | Causes a backslash to be printed |
| `\'` | Single quote | Causes a single quotation mark to be printed |
| `\"` | Double quote | Causes a double quotation mark to be printed |

> **WARNING!** Do not confuse the backslash (\) with the forward slash (/). An escape sequence will not work if you accidentally start it with a forward slash. Also, do not put a space between the backslash and the control character.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.7 The following program will not compile because the lines have been mixed up.

```
System.out.print("Success\n");
}
public class Success
{
System.out.print("Success\n");
public static void main(String[] args)
System.out.print("Success ");
}
// It's a mad, mad program.
System.out.println("\nSuccess");
{
```

When the lines are arranged properly, the program should display the following output on the screen:

**Program Output**

```
Success
Success Success

Success
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.8 Study the following program and show what it will print on the screen.

```
// The Works of Wolfgang
public class Wolfgang
{
   public static void main(String[] args)
   {
      System.out.print("The works of Wolfgang\ninclude ");
      System.out.print("the following");
      System.out.print("\nThe Turkish March ");
      System.out.print("and Symphony No. 40 ");
      System.out.println("in G minor.");
   }
}
```

2.9 On paper, write a program that will display your name on the first line; your street address on the second line; your city, state, and ZIP code on the third line; and your telephone number on the fourth line. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

## 2.3 Variables and Literals

**CONCEPT: A variable is a named storage location in the computer's memory. A literal is a value that is written into the code of a program.**

As you discovered in Chapter 1, variables allow you to store and work with data in the computer's memory. Part of the job of programming is to determine how many variables a program will need and what types of data they will hold. The program in Code Listing 2-7 is an example of a Java program with a variable.

**Code Listing 2-7** `(Variable.java)`

```
 1  // This program has a variable.
 2
 3  public class Variable
 4  {
 5     public static void main(String[] args)
 6     {
 7        int value;
 8
 9        value = 5;
10        System.out.print("The value is ");
11        System.out.println(value);
12     }
13  }
```

**Program Output**

```
The value is 5
```

Let's look more closely at this program. Here is line 7:

```
int value;
```

**VideoNote**
Declaring Variables

This is called a *variable declaration*. Variables must be declared before they can be used. A variable declaration tells the compiler the variable's name and the type of data it will hold. This line indicates the variable's name is value. The word `int` stands for integer, so value will only be used to hold integer numbers. Notice that variable declarations end with a semicolon. The next statement in this program appears in line 9:

```
value = 5;
```

This is called an *assignment statement*. The equal sign is an operator that stores the value on its right (in this case 5) into the variable named on its left. After this line executes, the value variable will contain the value 5.

**NOTE:** This line does not print anything on the computer screen. It runs silently behind the scenes.

Now look at lines 10 and 11:

```
System.out.print("The value is ");
System.out.println(value);
```

The statement in line 10 sends the string literal “The value is ” to the `print` method. The statement in line 11 sends the name of the value variable to the `println` method. When you send a variable name to `print` or `println`, the variable’s contents are displayed. Notice there are no quotation marks around `value`. Look at what happens in Code Listing 2-8.

**Code Listing 2-8** `(Variable2.java)`

```
// This program has a variable.

public class Variable2
{
   public static void main(String[] args)
   {
      int value;

      value = 5;
      System.out.print("The value is ");
      System.out.println("value");
   }
}
```

**Program Output**

```
The value is value
```

When double quotation marks are placed around the word `value` it becomes a string literal, not a variable name. When string literals are sent to `print` or `println`, they are displayed exactly as they appear inside the quotation marks.

## Displaying Multiple Items with the + Operator

When the + operator is used with strings, it is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
System.out.println("This is " + "one string.");
```

This statement will print:

```
This is one string.
```

The + operator produces a string that is the combination of the two strings used as its operands. You can also use the + operator to concatenate the contents of a variable to a string. The following code shows an example:

```
number = 5;
System.out.println("The value is " + number);
```

The second line uses the + operator to concatenate the contents of the `number` variable with the string “The value is ”. Although `number` is not a string, the + operator converts its value to a string and then concatenates that value with the first string. The output that will be displayed is:

```
The value is 5
```

Sometimes the argument you use with `print` or `println` is too long to fit on one line in your program code. However, a string literal cannot begin on one line and end on another. For example, the following will cause an error:

```
// This is an error!
System.out.println("Enter a value that is greater than zero
                    and less than 10." );
```

You can remedy this problem by breaking the argument up into smaller string literals, and then using the string concatenation operator to spread them out over more than one line. Here is an example:

```
System.out.println("Enter a value that is " +
                    "greater than zero and less " +
                    "than 10." );
```

In this statement, the argument is broken up into three strings and joined using the + operator. The following example shows the same technique used when the contents of a variable are part of the concatenation:

```
sum = 249;
System.out.println("The sum of the three " +
                    "numbers is " + sum);
```

## Be Careful with Quotation Marks

As shown in Code Listing 2-8, placing quotation marks around a variable name changes the program’s results. In fact, placing double quotation marks around anything that is not intended to be a string literal will create an error of some type. For example, in Code Listings 2-7 and 2-8, the number 5 was assigned to the variable `value`. It would have been an error to perform the assignment this way:

```
value = "5";  // Error!
```

In this statement, 5 is no longer an integer, but a string literal. Because `value` was declared as an integer variable, you can only store integers in it. In other words, 5 and “5” are not the same thing.

The fact that numbers can be represented as strings frequently confuses students who are new to programming. Just remember that strings are intended for humans to read. They are to be printed on computer screens or paper. Numbers, however, are intended primarily for mathematical operations. You cannot perform math on strings, and before numbers can be displayed on the screen, first they must be converted to strings. (Fortunately, `print` and `println` handle the conversion automatically when you send numbers to them.) Don’t fret if this still bothers you. Later in this chapter, we will shed more light on the differences among numbers, characters, and strings by discussing their internal storage.

## More about Literals

A literal is a value that is written in the code of a program. Literals are commonly assigned to variables or displayed. Code Listing 2-9 contains both literals and a variable.

**Code Listing 2-9** `(Literals.java)`

```java
// This program has literals and a variable.

public class Literals
{
   public static void main(String[] args)
   {
      int apples;

      apples = 20;
      System.out.println("Today we sold " + apples +
                         " bushels of apples.");
   }
}
```

**Program Output**

```
Today we sold 20 bushels of apples.
```

Of course, the variable in this program is `apples`. It is declared as an integer. Table 2-3 shows a list of the literals found in the program.

**Table 2-3** Literals

| Literal | Type of Literal |
|---|---|
| 20 | Integer literal |
| "Today we sold " | String literal |
| " bushels of apples." | String literal |

## Identifiers

An *identifier* is a programmer-defined name that represents some element of a program. Variable names and class names are examples of identifiers. You may choose your own variable names and class names in Java, as long as you do not use any of the Java key words. The *key words* make up the core of the language and each has a specific purpose. Table 1-3 in Chapter 1 and Appendix C (available on the book's companion Web site) show a complete list of Java key words.

You should always choose names for your variables that give an indication of what they are used for. You may be tempted to declare variables with names like this:

```java
int x;
```

The rather nondescript name, `x`, gives no clue as to what the variable's purpose is. Here is a better example.

```
int itemsOrdered;
```

The name `itemsOrdered` gives anyone reading the program an idea of what the variable is used for. This method of coding helps produce *self-documenting programs*, which means you get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines of code, it is important that they be as self-documenting as possible.

You have probably noticed the mixture of uppercase and lowercase letters in the name `itemsOrdered`. Although all of Java's key words must be written in lowercase, you may use uppercase letters in variable names. The reason the `O` in `itemsOrdered` is capitalized is to improve readability. Normally "items ordered" is used as two words. Variable names cannot contain spaces, however, so the two words must be combined. When "items" and "ordered" are stuck together, you get a variable declaration like this:

```
int itemsordered;
```

Capitalization of the letter `O` makes `itemsOrdered` easier to read. Typically, variable names begin with a lowercase letter, and after that, the first letter of each individual word that makes up the variable name is capitalized.

The following are some specific rules that must be followed with all identifiers:

- The first character must be one of the letters a–z or A–Z, an underscore (_), or a dollar sign ($).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (_), or dollar signs ($).
- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Identifiers cannot include spaces.

> **NOTE:** Although the $ is a legal identifier character, it is normally used for special purposes. So, don't use it in your variable names.

Table 2-4 shows a list of variable names and tells whether each is legal or illegal in Java.

**Table 2-4** Some variable names

| Variable Name | Legal or Illegal? |
|---|---|
| `dayOfWeek` | Legal |
| `3dGraph` | Illegal because identifiers cannot begin with a digit |
| `june1997` | Legal |
| `mixture#3` | Illegal because identifiers may use only alphabetic letters, digits, underscores, or dollar signs |
| `week day` | Illegal because identifiers cannot contain spaces |

## Class Names

As mentioned before, it is standard practice to begin variable names with a lowercase letter, and then capitalize the first letter of each subsequent word that makes up the name. It is also a standard practice to capitalize the first letter of a class name, as well as the first letter of each subsequent word it contains. This helps differentiate the names of variables from the names of classes. For example, `payRate` would be a variable name, and `Employee` would be a class name.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.10 Examine the following program.

```
// This program uses variables and literals.
public class BigLittle
{
   public static void main(String[] args)
   {
      int little;
      int big;
      little = 2;
      big = 2000;
      System.out.println("The little number is " + little);
      System.out.println("The big number is " + big);
   }
}
```

List the variables and literals found in the program.

2.11 What will the following program display on the screen?

```
public class CheckPoint
{
   public static void main(String[] args)
   {
      int number;
      number = 712;
      System.out.println("The value is " + "number");
   }
}
```

## 2.4 Primitive Data Types

**CONCEPT: There are many different types of data. Variables are classified according to their data type, which determines the kind of data that may be stored in them.**

Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric data, for example, there are whole and fractional numbers, negative and positive numbers, and numbers so large and others so small that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as strings of characters. When you write a program you must determine what types of data it is likely to encounter.

Each variable has a *data type*, which is the type of data that the variable can hold. Selecting the proper data type is important because a variable's data type determines the amount of memory the variable uses, and the way the variable formats and stores data. It is important to select a data type that is appropriate for the type of data that your program will work with. If you are writing a program to calculate the number of miles to a distant star, you need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you need variables that store very small and precise numbers. If you are writing a program that must perform thousands of intensive calculations, you want variables that can be processed quickly. The data type of a variable determines all of these factors.

Table 2-5 shows all of the Java *primitive data types* for holding numeric data.

The words listed in the left column of Table 2-5 are the key words that you use in variable declarations. A variable declaration takes the following general format:

```
DataType VariableName;
```

**Table 2-5** Primitive data types for numeric data

| Data Type | Size | Range |
|---|---|---|
| `byte` | 1 byte | Integers in the range of −128 to +127 |
| `short` | 2 bytes | Integers in the range of −32,768 to +32,767 |
| `int` | 4 bytes | Integers in the range of −2,147,483,648 to +2,147,483,647 |
| `long` | 8 bytes | Integers in the range of −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 |
| `float` | 4 bytes | Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy |
| `double` | 8 bytes | Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy |

*`DataType`* is the name of the data type and *`VariableName`* is the name of the variable. Here are some examples of variable declarations:

```
byte inches;
int speed;
short month;
float salesCommission;
double distance;
```

The size column in Table 2-5 shows the number of bytes that a variable of each of the data types uses. For example, an `int` variable uses 4 bytes, and a `double` variable uses 8 bytes.

The range column shows the ranges of numbers that may be stored in variables of each data type. For example, an `int` variable can hold numbers from −2,147,483,648 up to +2,147,483,647. One of the appealing characteristics of the Java language is that the sizes and ranges of all the primitive data types are the same on all computers.

> **NOTE:** These data types are called "primitive" because you cannot use them to create objects. Recall from Chapter 1's discussion on object-oriented programming that an object has attributes and methods. With the primitive data types, you can only create variables, and a variable can only be used to hold a single value. Such variables do not have attributes or methods.

## The Integer Data Types

The first four data types listed in Table 2-5, `byte`, `int`, `short`, and `long`, are all integer data types. An integer variable can hold whole numbers such as 7, 125, −14, and 6928. The program in Code Listing 2-10 shows several variables of different integer data types being used.

**Code Listing 2-10** `(IntegerVariables.java)`

```
// This program has variables of several of the integer types.

public class IntegerVariables
{
   public static void main(String[] args)
   {
      int checking; // Declare an int variable named checking.
      byte miles; // Declare a byte variable named miles.
      short minutes; // Declare a short variable named minutes.
      long days; // Declare a long variable named days.

      checking = -20;
      miles = 105;
      minutes = 120;
      days = 189000;
      System.out.println("We have made a journey of " + miles +
                         " miles.");
      System.out.println("It took us " + minutes + " minutes.");
      System.out.println("Our account balance is $" + checking);
      System.out.println("About " + days + " days ago Columbus " +
                         "stood on this spot.");
   }
}
```

**Program Output**

```
We have made a journey of 105 miles.
It took us 120 minutes.
Our account balance is $-20
About 189000 days ago Columbus stood on this spot.
```

In most programs you will need more than one variable of any given data type. If a program uses three integers, `length`, `width`, and `area`, they could be declared separately, as follows:

```
int length;
int width;
int area;
```

It is easier, however, to combine the three variable declarations:

```
int length, width, area;
```

You can declare several variables of the same type, simply by separating their names with commas.

### Integer Literals

When you write an integer literal in your program code, Java assumes it to be of the `int` data type. For example, in Code Listing 2-10, the literals −20, 105, 120, and 189000 are all treated as `int` values. You can force an integer literal to be treated as a long, however, by suffixing it with the letter L. For example, the value 57L would be treated as a long. Although you can use either an uppercase or a lowercase L, it is advisable to use the uppercase L because the lowercase l looks too much like the number 1.

**WARNING!** You cannot embed commas in numeric literals. For example, the following statement will cause an error:

```
number = 1,257,649;     // ERROR!
```

This statement must be written as:

```
number = 1257649;       // Correct.
```

## Floating-Point Data Types

Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers. Values such as 1.7 and −45.316 are floating-point numbers.

In Java there are two data types that can represent floating-point numbers. They are `float` and `double`. The `float` data type is considered a single precision data type. It can store a floating-point number with 7 digits of accuracy. The `double` data type is considered a double precision data type. It can store a floating-point number with 15 digits of accuracy. The `double` data type uses twice as much memory as the `float` data type, however. A `float` variable occupies 4 bytes of memory, whereas a `double` variable uses 8 bytes.

Code Listing 2-11 shows a program that uses three `double` variables.

**Code Listing 2-11** `(Sale.java)`

```
// This program demonstrates the double data type.

public class Sale
{
   public static void main(String[] args)
   {
      double price, tax, total;

      price = 29.75;
      tax = 1.76;
      total = 31.51;
      System.out.println("The price of the item " +
                         "is " + price);
      System.out.println("The tax is " + tax);
      System.out.println("The total is " + total);
   }
}
```

**Program Output**

```
The price of the item is 29.75
The tax is 1.76
The total is 31.51
```

### Floating-Point Literals

When you write a floating-point literal in your program code, Java assumes it to be of the `double` data type. For example, in Code Listing 2-11, the literals 29.75, 1.76, and 31.51 are all treated as `double` values. Because of this, a problem can arise when assigning a floating-point literal to a `float` variable. Java is a *strongly typed language*, which means that it only allows you to store values of compatible data types in variables. A `double` value is not compatible with a `float` variable because a `double` can be much larger or much smaller than the allowable range for a `float`. As a result, code such as the following will cause an error:

```
float number;
number = 23.5;            // Error!
```

You can force a `double` literal to be treated as a `float`, however, by suffixing it with the letter F or f. The preceding code can be rewritten in the following manner to prevent an error:

```
float number;
number = 23.5F;            // This will work.
```

**WARNING!** If you are working with literals that represent dollar amounts, remember that you cannot embed currency symbols (such as $) or commas in the literal. For example, the following statement will cause an error:

```
grossPay = $1,257.00;       // ERROR!
```

This statement must be written as:

```
grossPay = 1257.00;         // Correct.
```

### Scientific and E Notation

Floating-point literals can be represented in scientific notation. Take the number 47,281.97. In scientific notation this number is $4.728197 \times 10^4$. ($10^4$ is equal to 10,000, and $4.728197 \times 10{,}000$ is 47,281.97.)

Java uses E notation to represent values in scientific notation. In E notation, the number $4.728197 \times 10^4$ would be 4.728197E4. Table 2-6 shows other numbers represented in scientific and E notation.

**Table 2-6** Floating-point representations

| Decimal Notation | Scientific Notation | E Notation |
|---|---|---|
| 247.91 | $2.4791 \times 10^2$ | 2.4791E2 |
| 0.00072 | $7.2 \times 10^{-4}$ | 7.2E–4 |
| 2,900,000 | $2.9 \times 10^6$ | 2.9E6 |

**NOTE:** The E can be uppercase or lowercase.

Code Listing 2-12 demonstrates the use of floating-point literals expressed in E notation.

**Code Listing 2-12** (`SunFacts.java`)

```java
// This program uses E notation.

public class SunFacts
{
   public static void main(String[] args)
   {
      double distance, mass;

      distance = 1.495979E11;
      mass = 1.989E30;
      System.out.println("The sun is " + distance +
                         " meters away.");
      System.out.println("The sun's mass is " + mass +
                         " kilograms.");
   }
}
```

**Program Output**

```
The sun is 1.495979E11 meters away.
The sun's mass is 1.989E30 kilograms.
```

## The `boolean` Data Type

The `boolean` data type allows you to create variables that may hold one of two possible values: `true` or `false`. Code Listing 2-13 demonstrates the declaration and assignment of a `boolean` variable.

**Code Listing 2-13** `(TrueFalse.java)`

```
// A program for demonstrating boolean variables

public class TrueFalse
{
   public static void main(String[] args)
   {
      boolean bool;

      bool = true;
      System.out.println(bool);
      bool = false;
      System.out.println(bool);
   }
}
```

**Program Output**

```
true
false
```

Variables of the `boolean` data type are useful for evaluating conditions that are either true or false. You will not be using them until Chapter 3, however, so for now just remember the following things:

- `boolean` variables may hold only the value `true` or `false`.
- The contents of a `boolean` variable may not be copied to a variable of any type other than `boolean`.

## The `char` Data Type

The `char` data type is used to store characters. A variable of the `char` data type can hold one character at a time. Character literals are enclosed in *single quotation marks*. The program in Code Listing 2-14 uses a `char` variable. The character literals ‘A’ and ‘B’ are assigned to the variable.

**Code Listing 2-14** `(Letters.java)`

```
// This program demonstrates the char data type.

public class Letters
{
   public static void main(String[] args)
   {
      char letter;

      letter = 'A';
      System.out.println(letter);
      letter = 'B';
      System.out.println(letter);
   }
}
```

**Program Output**

```
A
B
```

It is important that you do not confuse character literals with string literals, which are enclosed in double quotation marks. String literals cannot be assigned to `char` variables.

### Unicode

Characters are internally represented by numbers. Each printable character, as well as many non-printable characters, is assigned a unique number. Java uses Unicode, which is a set of numbers that are used as codes for representing characters. Each Unicode number requires two bytes of memory, so `char` variables occupy two bytes. When a character is stored in memory, it is actually the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.

You may want to refer to Appendix A, available on the book's companion Web site (at www.pearsonhighered.com/gaddis), which shows a portion of the Unicode character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. Code Listing 2-15 demonstrates that when you work with characters, you are actually working with numbers.

**Code Listing 2-15** `(Letters2.java)`

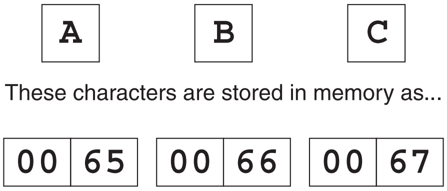
```
// This program demonstrates the close relationship between
// characters and integers.

public class Letters2
{
   public static void main(String[] args)
   {
      char letter;

      letter = 65;
      System.out.println(letter);
      letter = 66;
      System.out.println(letter);
   }
}
```

**Program Output**

```
A
B
```

Figure 2-4 illustrates that when you think of the characters A, B, and C being stored in memory, it is really the numbers 65, 66, and 67 that are stored.

**Figure 2-4** Characters and how they are stored in memory



## Variable Assignment and Initialization

As you have already seen in several examples, a value is put into a variable with an *assignment statement*. For example, the following statement assigns the value 12 to the variable `unitsSold`:

```
unitsSold = 12;
```

The = symbol is called the assignment operator. Operators perform operations on data. The data that operators work with are called operands. The assignment operator has two operands. In the statement above, the operands are `unitsSold` and `12`.

In an assignment statement, the name of the variable receiving the assignment must appear on the left side of the operator, and the value being assigned must appear on the right side. The following statement is incorrect:

```
12 = unitsSold;      // ERROR!
```

The operand on the left side of the = operator must be a variable name. The operand on the right side of the = symbol must be an expression that has a value. The assignment operator takes the value of the right operand and puts it in the variable identified by the left operand. Assuming that `length` and `width` are both `int` variables, the following code illustrates that the assignment operator's right operand may be a literal or a variable:

```
length = 20;
width = length;
```

It is important to note that the assignment operator only changes the contents of its left operand. The second statement assigns the value of the `length` variable to the `width` variable. After the statement has executed, `length` still has the same value, 20.

You may also assign values to variables as part of the declaration statement. This is known as *initialization*. Code Listing 2-16 shows how it is done.

The variable declaration statement in this program is in line 7:

```
int month = 2, days = 28;
```

**Code Listing 2-16** `(Initialize.java)`

```java
// This program shows variable initialization.

public class Initialize
{
   public static void main(String[] args)
   {
      int month = 2, days = 28;

      System.out.println("Month " + month + " has " +
                         days + " days.");
   }
}
```

**Program Output**

```
Month 2 has 28 days.
```

This statement declares the `month` variable and initializes it with the value 2, and declares the `days` variable and initializes it with the value 28. As you can see, this simplifies the program and reduces the number of statements that must be typed by the programmer. Here are examples of other declaration statements that perform initialization:

```java
double payRate = 25.52;
float interestRate = 12.9F;
char stockCode = 'D';
int customerNum = 459;
```

Of course, there are always variations on a theme. Java allows you to declare several variables and initialize only some of them. Here is an example of such a declaration:

```java
int flightNum = 89, travelTime, departure = 10, distance;
```

The variable `flightNum` is initialized to 89 and `departure` is initialized to 10. The `travelTime` and `distance` variables remain uninitialized.

**WARNING!** When a variable is declared inside a method, it must have a value stored in it before it can be used. If the compiler determines that the program might be using such a variable before a value has been stored in it, an error will occur. You can avoid this type of error by initializing the variable with a value.

## Variables Hold Only One Value at a Time

Remember, a variable can hold only one value at a time. When you assign a new value to a variable, the new value takes the place of the variable's previous contents. For example, look at the following code.

```java
int x = 5;
System.out.println(x);
x = 99;
System.out.println(x);
```

In this code, the variable `x` is initialized with the value 5 and its contents are displayed. Then the variable is assigned the value 99. This value overwrites the value 5 that was previously stored there. The code will produce the following output:

```
5
99
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.12 Which of the following are illegal variable names and why?

```
x
99bottles
july97
theSalesFigureForFiscalYear98
r&d
grade_report
```

2.13 Is the variable name `Sales` the same as `sales`? Why or why not?

2.14 Refer to the Java primitive data types listed in Table 2-5 for this question.
  a) If a variable needs to hold whole numbers in the range 32 to 6,000, what primitive data type would be best?
  b) If a variable needs to hold whole numbers in the range −40,000 to +40,000, what primitive data type would be best?
  c) Which of the following literals use more memory? `22.1` or `22.1F`?

2.15 How would the number $6.31 \times 10^{17}$ be represented in E notation?

2.16 A program declares a `float` variable named `number`, and the following statement causes an error. What can be done to fix the error?

```
number = 7.4;
```

2.17 What values can `boolean` variables hold?

2.18 Write statements that do the following:
  a) Declare a `char` variable named `letter`.
  b) Assign the letter A to the `letter` variable.
  c) Display the contents of the `letter` variable.

2.19 What are the Unicode codes for the characters 'C', 'F', and 'W'? (You may need to refer to Appendix A on the book's companion Web site, at www.pearsonhighered.com/gaddis.)

2.20 Which is a character literal, `'B'` or `"B"`?

2.21 What is wrong with the following statement?

```
char letter = "Z";
```

## 2.5 Arithmetic Operators

**CONCEPT: There are many operators for manipulating numeric values and performing arithmetic operations.**

Java offers a multitude of operators for manipulating data. Generally, there are three types of operators: *unary*, *binary*, and *ternary*. These terms reflect the number of operands an operator requires.

VideoNote
Simple Math Expressions

Unary operators require only a single operand. For example, consider the following expression:

```
-5
```

Of course, we understand this represents the value negative five. We can also apply the operator to a variable, as follows:

```
-number
```

This expression gives the negative of the value stored in `number`. The minus sign, when used this way, is called the *negation operator*. Because it requires only one operand, it is a unary operator.

Binary operators work with two operands. The assignment operator is in this category. Ternary operators, as you may have guessed, require three operands. Java has only one ternary operator, which is discussed in Chapter 3.

Arithmetic operations are very common in programming. Table 2-7 shows the arithmetic operators in Java.

**Table 2-7** Arithmetic operators

| Operator | Meaning | Type | Example |
|---|---|---|---|
| `+` | Addition | Binary | `total = cost + tax;` |
| `-` | Subtraction | Binary | `cost = total - tax;` |
| `*` | Multiplication | Binary | `tax = cost * rate;` |
| `/` | Division | Binary | `salePrice = original / 2;` |
| `%` | Modulus | Binary | `remainder = value % 3;` |

Each of these operators works as you probably expect. The addition operator returns the sum of its two operands. Here are some example statements that use the addition operator:

```
amount = 4 + 8;              // Assigns 12 to amount
total = price + tax;         // Assigns price + tax to total
number = number + 1;         // Assigns number + 1 to number
```

The subtraction operator returns the value of its right operand subtracted from its left operand. Here are some examples:

```
temperature = 112 - 14;      // Assigns 98 to temperature
sale = price - discount;     // Assigns price - discount to sale
number = number - 1;         // Assigns number - 1 to number
```

The multiplication operator returns the product of its two operands. Here are some examples:

```
markUp = 12 * 0.25;          // Assigns 3 to markUp
commission = sales * percent; // Assigns sales * percent to commission
population = population * 2;  // Assigns population * 2 to population
```

The division operator returns the quotient of its left operand divided by its right operand. Here are some examples:

```
points = 100 / 20;                // Assigns 5 to points
teams = players / maxEach;        // Assigns players / maxEach to teams
half = number / 2;                // Assigns number / 2 to half
```

The modulus operator returns the remainder of a division operation involving two integers. The following statement assigns 2 to `leftOver`:

```
leftOver = 17 % 3;
```

Situations arise where you need to get the remainder of a division. Computations that detect odd numbers or are required to determine how many items are left over after division use the modulus operator.

The program in Code Listing 2-17 demonstrates some of these operators used in a simple payroll calculation.

### Code Listing 2-17 (Wages.java)

```
// This program calculates hourly wages plus overtime.

public class Wages
{
   public static void main(String[] args)
   {
      double regularWages;       // The calculated regular wages.
      double basePay = 25;       // The base pay rate.
      double regularHours = 40;  // The hours worked less overtime.
      double overtimeWages;      // Overtime wages
      double overtimePay = 37.5; // Overtime pay rate
      double overtimeHours = 10; // Overtime hours worked
      double totalWages;         // Total wages

      regularWages = basePay * regularHours;
      overtimeWages = overtimePay * overtimeHours;
      totalWages = regularWages + overtimeWages;
      System.out.println("Wages for this week are $" +
                         totalWages);
   }
}
```

**Program Output**

```
Wages for this week are $1375.0
```

Code Listing 2-17 calculates the total wages an hourly paid worker earned in one week. As mentioned in the comments, there are variables for regular wages, base pay rate, regular hours worked, overtime wages, overtime pay rate, overtime hours worked, and total wages.

Line 15 in the program multiplies `basePay` times `regularHours` and stores the result, which is 1000, in `regularWages`:

```
regularWages = basePay * regularHours;
```

Line 16 multiplies `overtimePay` times `overtimeHours` and stores the result, which is 375, in `overtimeWages`:

```
overtimeWages = overtimePay * overtimeHours;
```

Line 17 adds the regular wages and the overtime wages and stores the result, 1375, in `totalWages`:

```
totalWages = regularWages + overtimeWages;
```

The `println` statement in lines 18 and 19 displays the message on the screen reporting the week's wages.

## Integer Division

When both operands of the division operator are integers, the operator will perform *integer division*. This means the result of the division will be an integer as well. If there is a remainder, it will be discarded. For example, look at the following code:

```
double number;
number = 5 / 2;
```

This code divides 5 by 2 and assigns the result to the `number` variable. What value will be stored in `number`? You would probably assume that 2.5 would be stored in `number` because that is the result your calculator shows when you divide 5 by 2; however, that is not what happens when the previous Java code is executed. Because the numbers 5 and 2 are both integers, the fractional part of the result will be thrown away, or *truncated*. As a result, the value 2 will be assigned to the `number` variable.

In the previous code, it doesn't matter that `number` is declared as a `double` because the fractional part of the result is discarded before the assignment takes place. In order for a division operation to return a floating-point value, one of the operands must be of a floating-point data type. For example, the previous code could be written as follows:

```
double number;
number = 5.0 / 2;
```

In this code, 5.0 is treated as a floating-point number, so the division operation will return a floating-point number. The result of the division is 2.5.

## Operator Precedence

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, `x`, 21, and `y` to the variable `answer`:

```
answer = 17 + x + 21 + y;
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12 + 6 / 3;
```

What value will be stored in `outcome`? The 6 is used as an operand for both the addition and division operators. The `outcome` variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the division operator has higher *precedence* than the addition operator.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the operator with the highest precedence works first. Multiplication and division have higher precedence than addition and subtraction, so the statement above works like this:

1. 6 is divided by 3, yielding a result of 2
2. 12 is added to 2, yielding a result of 14

It could be diagrammed as shown in Figure 2-5.

**Figure 2-5** Precedence illustrated

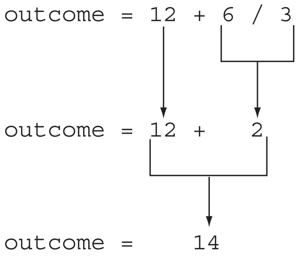


Table 2-8 shows the precedence of the arithmetic operators. The operators at the top of the table have higher precedence than the ones below them.

**Table 2-8** Precedence of arithmetic operators (highest to lowest)

| | |
|---|---|
| **Highest Precedence** → | – (unary negation) |
| | `* / %` |
| **Lowest Precedence** → | `+ -` |

The multiplication, division, and modulus operators have the same precedence. The addition and subtraction operators have the same precedence. If two operators sharing an operand have the same precedence, they work according to their *associativity*. Associativity is either *left to right* or *right to left*. Table 2-9 shows the arithmetic operators and their associativity.

**Table 2-9** Associativity of arithmetic operators

| Operator | Associativity |
|---|---|
| – (unary negation) | Right to left |
| `* / %` | Left to right |
| `+ -` | Left to right |

Table 2-10 shows some expressions and their values.

**Table 2-10** Some expressions and their values

| Expression | Value |
|---|---|
| `5 + 2 * 4` | 13 |
| `10 / 2 - 3` | 2 |
| `8 + 12 * 2 - 4` | 28 |
| `4 + 17 % 2 - 1` | 4 |
| `6 - 3 * 2 + 7 - 1` | 6 |

## Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the statement below, the sum of `a`, `b`, `c`, and `d` is divided by `4.0`.

```
average = (a + b + c + d) / 4.0;
```

Without the parentheses, however, `d` would be divided by `4` and the result added to `a`, `b`, and `c`. Table 2-11 shows more expressions and their values.

**Table 2-11** More expressions and their values

| Expression | Value |
|---|---|
| `(5 + 2) * 4` | 28 |
| `10 / (5 - 3)` | 5 |
| `8 + 12 * (6 - 2)` | 56 |
| `(4 + 17) % 2 - 1` | 0 |
| `(6 - 3) * (2 + 7) / 3` | 9 |

## In the Spotlight:

### Calculating Percentages and Discounts

Determining percentages is a common calculation in computer programming. Although the % symbol is used in general mathematics to indicate a percentage, most programming languages (including Java) do not use the % symbol for this purpose. In a program, you have to convert a percentage to a floating-point number, just as you would if you were using a calculator. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's look at an example. Suppose you earn $6,000 per month and you are allowed to contribute a portion of your gross monthly pay to a retirement plan. You want to determine the amount of your pay that will go into the plan if you contribute 5 percent, 8 percent, or 10 percent of your gross wages. To make this determination you write a program like the one shown in Code Listing 2-18.

**Code Listing 2-18** `(Contribution.java)`

```
 1  // This program calculates the amount of pay that
 2  // will be contributed to a retirement plan if 5%,
 3  // 8%, or 10% of monthly pay is withheld.
 4
 5  public class Contribution
 6  {
 7     public static void main(String[] args)
 8    {
 9       // Variables to hold the monthly pay and
10       // the amount of contribution.
11       double monthlyPay = 6000.0;
12       double contribution;
13
14       // Calculate and display a 5% contribution.
15       contribution = monthlyPay * 0.05;
16       System.out.println("5 percent is $" +
17                          contribution +
18                          " per month.");
19
20       // Calculate and display an 8% contribution.
21       contribution = monthlyPay * 0.08;
22       System.out.println("8 percent is $" +
23                          contribution +
24                          " per month.");
25
26       // Calculate and display a 10% contribution.
27       contribution = monthlyPay * 0.1;
28       System.out.println("10 percent is $" +
29                          contribution +
30                          " per month.");
31     }
32  }
```

**Program Output**

```
5 percent is $300.0 per month.
8 percent is $480.0 per month.
10 percent is $600.0 per month.
```

Lines 11 and 12 declare two variables: `monthlyPay` and `contribution`. The `monthlyPay` variable, which is initialized with the value `6000.0`, holds the amount of your monthly pay. The `contribution` variable will hold the amount of a contribution to the retirement plan.

The statements in lines 15 through 18 calculate and display 5 percent of the monthly pay. The calculation is done in line 15, where the `monthlyPay` variable is multiplied by 0.05. The result is assigned to the `contribution` variable, which is then displayed by the statement in lines 16 through 18.

Similar steps are taken in lines 21 through 24, which calculate and display 8 percent of the monthly pay, and lines 27 through 30, which calculate and display 10 percent of the monthly pay.

### Calculating a Percentage Discount

Another common calculation is determining a percentage discount. For example, suppose a retail business sells an item that is regularly priced at $59, and is planning to have a sale where the item's price will be reduced by 20 percent. You have been asked to write a program to calculate the sale price of the item.

To determine the sale price you perform two calculations:

- First, you get the amount of the discount, which is 20 percent of the item's regular price.
- Second, you subtract the discount amount from the item's regular price. This gives you the sale price.

Code Listing 2-19 shows how this is done in Java.

**Code Listing 2-19** `(Discount.java)`

```
// This program calculates the sale price of an
// item that is regularly priced at $59, with
// a 20 percent discount subtracted.

public class Discount
{
   public static void main(String[] args)
   {
      // Variables to hold the regular price, the
      // amount of a discount, and the sale price.
      double regularPrice = 59.0;
      double discount;
      double salePrice;

      // Calculate the amount of a 20% discount.
      discount = regularPrice * 0.2;

      // Calculate the sale price by subtracting
      // the discount from the regular price.
      salePrice = regularPrice - discount;

      // Display the results.
      System.out.println("Regular price: $" + regularPrice);
      System.out.println("Discount amount $" + discount);
      System.out.println("Sale price: $" + salePrice);
   }
}
```

**Program Output**

```
Regular price: $59.0
Discount amount $11.8
Sale price: $47.2
```

Lines 11 through 13 declare three variables. The `regularPrice` variable holds the item's regular price, and is initialized with the value `59.0`. The `discount` variable will hold the amount of the discount once it is calculated. The `salePrice` variable will hold the item's sale price.

Line 16 calculates the amount of the 20 percent discount by multiplying `regularPrice` by 0.2. The result is stored in the `discount` variable. Line 20 calculates the sale price by subtracting `discount` from `regularPrice`. The result is stored in the `salePrice` variable. The statements in lines 23 through 25 display the item's regular price, the amount of the discount, and the sale price.

## The `Math` Class

The Java API provides a class named `Math`, which contains numerous methods that are useful for performing complex mathematical operations. In this section we will briefly look at the `Math.pow` and `Math.sqrt` methods.

### The `Math.pow` Method

In Java, raising a number to a power requires the `Math.pow` method. Here is an example of how the `Math.pow` method is used:

```
result = Math.pow(4.0, 2.0);
```

The method takes two `double` arguments. It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

$$result = 4^2$$

Here is another example of a statement using the `Math.pow` method. It assigns 3 times $6^3$ to `x`:

```
x = 3 * Math.pow(6.0, 3.0);
```

And the following statement displays the value of 5 raised to the power of 4:

```
System.out.println(Math.pow(5.0, 4.0));
```

### The `Math.sqrt` Method

The `Math.sqrt` method accepts a `double` value as its argument and returns the square root of the value. Here is an example of how the method is used:

```
result = Math.sqrt(9.0);
```

In this example the value 9.0 is passed as an argument to the `Math.sqrt` method. The method will return the square root of 9.0, which is assigned to the `result` variable. The following statement shows another example. In this statement the square root of 25.0 (which is 5.0) is displayed on the screen:

```
System.out.println(Math.sqrt(25.0));
```

**NOTE:** For more information about the `Math` class, see Appendix F, available on the book's companion Web site at www.pearsonhighered.com/gaddis.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

2.22 Complete the following table by writing the value of each expression in the Value column.

| Expression | Value |
|---|---|
| `6 + 3 * 5` | ______ |
| `12 / 2 - 4` | ______ |
| `9 + 14 * 2 - 6` | ______ |
| `5 + 19 % 3 - 1` | ______ |
| `(6 + 2) * 3` | ______ |
| `14 / (11 - 4)` | ______ |
| `9 + 12 * (8 - 3)` | ______ |

2.23 Is the division statement in the following code an example of integer division or floating-point division? What value will be stored in `portion`?

```
double portion;
portion = 70 / 3;
```

## 2.6 Combined Assignment Operators

**CONCEPT: The combined assignment operators combine the assignment operator with the arithmetic operators.**

Quite often, programs have assignment statements of the following form:

```
x = x + 1;
```

On the right side of the assignment operator, 1 is added to `x`. The result is then assigned to `x`, replacing the value that was previously there. Effectively, this statement adds 1 to x. Here is another example:

```
balance = balance + deposit;
```

Assuming that `balance` and `deposit` are variables, this statement assigns the value of `balance + deposit` to `balance`. The effect of this statement is that `deposit` is added to the value stored in `balance`. Here is another example:

```
balance = balance - withdrawal;
```

Assuming that `balance` and `withdrawal` are variables, this statement assigns the value of `balance - withdrawal` to `balance`. The effect of this statement is that withdrawal is subtracted from the value stored in `balance`.

If you have not seen these types of statements before, they might cause some initial confusion because the same variable name appears on both sides of the assignment operator. Table 2-12 shows other examples of statements written this way.

**Table 2-12** Various assignment statements (assume x = 6 in each statement)

| Statement | What It Does | Value of x after the Statement |
|---|---|---|
| `x = x + 4;` | Adds 4 to `x` | 10 |
| `x = x - 3;` | Subtracts 3 from `x` | 3 |
| `x = x * 10;` | Multiplies `x` by 10 | 60 |
| `x = x / 2;` | Divides `x` by 2 | 3 |
| `x = x % 4` | Assigns the remainder of x / 4 to x. | 2 |

These types of operations are common in programming. For convenience, Java offers a special set of operators designed specifically for these jobs. Table 2-13 shows the *combined assignment operators*, also known as *compound operators*.

**Table 2-13** Combined assignment operators

| Operator | Example Usage | Equivalent To |
|---|---|---|
| `+=` | `x += 5;` | `x = x + 5;` |
| `-=` | `y -= 2;` | `y = y - 2;` |
| `*=` | `z *= 10;` | `z = z * 10;` |
| `/=` | `a /= b;` | `a = a / b;` |
| `%=` | `c %= 3;` | `c = c % 3;` |

As you can see, the combined assignment operators do not require the programmer to type the variable name twice. The following statement:

```
balance = balance + deposit;
```

could be rewritten as

```
balance += deposit;
```

Similarly, the statement

```
balance = balance - withdrawal;
```

could be rewritten as

```
balance -= withdrawal;
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

2.24 Write statements using combined assignment operators to perform the following:
- a) Add 6 to `x`
- b) Subtract 4 from `amount`

c) Multiply `y` by 4
d) Divide `total` by 27
e) Store in `x` the remainder of `x` divided by 7

## 2.7 Conversion between Primitive Data Types

**CONCEPT: Before a value can be stored in a variable, the value's data type must be compatible with the variable's data type. Java performs some conversions between data types automatically, but does not automatically perform any conversion that can result in the loss of data. Java also follows a set of rules when evaluating arithmetic expressions containing mixed data types.**

Java is a *strongly typed* language. This means that before a value is assigned to a variable, Java checks the data types of the variable and the value being assigned to it to determine whether they are compatible. For example, look at the following statements:

```
int x;
double y = 2.5;
x = y;
```

The assignment statement is attempting to store a `double` value (2.5) in an `int` variable. When the Java compiler encounters this line of code, it will respond with an error message. (The JDK displays the message "possible loss of precision.")

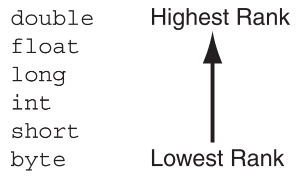
Not all assignment statements that mix data types are rejected by the compiler, however. For instance, look at the following program segment:

```
int x;
short y = 2;
x = y;
```

This assignment statement, which stores a `short` in an `int`, will work with no problems. So why does Java permit a `short` to be stored in an `int`, but does not permit a `double` to be stored in an `int`? The obvious reason is that a `double` can store fractional numbers and can hold values much larger than an `int` can hold. If Java were to permit a `double` to be assigned to an `int`, a loss of data would be likely.

Just like officers in the military, the primitive data types are ranked. One data type outranks another if it can hold a larger number. For example, a `float` outranks an `int`, and an `int` outranks a `short`. Figure 2-6 shows the numeric data types in order of their rank. The higher a data type appears in the list, the higher is its rank.

**Figure 2-6** Primitive data type ranking



In assignment statements where values of lower-ranked data types are stored in variables of higher-ranked data types, Java automatically converts the lower-ranked value to the higher-ranked type. This is called a *widening conversion*. For example, the following code demonstrates a widening conversion, which takes place when an `int` value is stored in a `double` variable:

```
double x;
int y = 10;
x = y;        // Performs a widening conversion
```

A *narrowing conversion* is the conversion of a value to a lower-ranked type. For example, converting a `double` to an `int` would be a narrowing conversion. Because narrowing conversions can potentially cause a loss of data, Java does not automatically perform them.

### Cast Operators

The *cast operator* lets you manually convert a value, even if it means that a narrowing conversion will take place. Cast operators are unary operators that appear as a data type name enclosed in a set of parentheses. The operator precedes the value being converted. Here is an example:

```
x = (int)number;
```

The cast operator in this statement is the word `int` inside the parentheses. It returns the value in `number`, converted to an `int`. This converted value is then stored in `x`. If number were a floating-point variable, such as a `float` or a `double`, the value that is returned would be *truncated*, which means the fractional part of the number is lost. The original value in the number variable is not changed, however.

Table 2-14 shows several statements using cast operators.

**Table 2-14** Example uses of cast operators

| Statement | Description |
|---|---|
| `littleNum = (short)bigNum;` | The cast operator returns the value in `bigNum`, converted to a `short`. The converted value is assigned to the variable `littleNum`. |
| `x = (long)3.7;` | The cast operator is applied to the expression 3.7. The operator returns the value 3, which is assigned to the variable `x`. |
| `number = (int)72.567;` | The cast operator is applied to the expression 72.567. The operator returns 72, which is used to initialize the variable `number`. |
| `value = (float)x;` | The cast operator returns the value in `x`, converted to a `float`. The converted value is assigned to the variable `value`. |
| `value = (byte)number;` | The cast operator returns the value in `number`, converted to a `byte`. The converted value is assigned to the variable `value`. |

Note that when a cast operator is applied to a variable, it does not change the contents of the variable. It only returns the value stored in the variable, converted to the specified data type.

Recall from our earlier discussion that when both operands of a division are integers, the operation will result in integer division. This means that the result of the division will be an integer, with any fractional part of the result thrown away. For example, look at the following code:

```
int pies = 10, people = 4;
double piesPerPerson;
piesPerPerson = pies / people;
```

Although 10 divided by 4 is 2.5, this code will store 2 in the `piesPerPerson` variable. Because both `pies` and `people` are `int` variables, the result will be an `int`, and the fractional part will be thrown away. We can modify the code with a cast operator, however, so it gives the correct result as a floating-point value:

```
piesPerPerson = (double)pies / people;
```

The variable `pies` is an `int` and holds the value 10. The expression `(double)pies` returns the value in `pies` converted to a `double`. This means that one of the division operator's operands is a `double`, so the result of the division will be a `double`. The statement could also have been written as follows:

```
piesPerPerson = pies / (double)people;
```

In this statement, the cast operator returns the value of the `people` variable converted to a `double`. In either statement, the result of the division is a `double`.

**WARNING!** The cast operator can be applied to an entire expression enclosed in parentheses. For example, look at the following statement:

```
piesPerPerson = (double)(pies / people);
```

This statement does not convert the value in `pies` or `people` to a `double`, but converts the result of the expression `pies / people`. If this statement were used, an integer division operation would still have been performed. Here's why: The result of the expression `pies / people` is 2 (because integer division takes place). The value 2 converted to a `double` is 2.0. To prevent the integer division from taking place, one of the operands must be converted to a `double`.

## Mixed Integer Operations

One of the nuances of the Java language is the way it internally handles arithmetic operations on `int`, `byte`, and `short` variables. When values of the `byte` or `short` data types are used in arithmetic expressions, they are temporarily converted to `int` values. The result of an arithmetic operation using only a mixture of `byte`, `short`, or `int` values will always be an `int`.

For example, assume that `b` and `c` in the following expression are `short` variables:

```
b + c
```

Although both `b` and `c` are `short` variables, the result of the expression `b + c` is an `int`. This means that when the result of such an expression is stored in a variable, the variable must be an `int` or higher data type. For example, look at the following code:

```
short firstNumber = 10,
      secondNumber = 20,
      thirdNumber;
// The following statement causes an error!
thirdNumber = firstNumber + secondNumber;
```

When this code is compiled, the following statement causes an error:

```
thirdNumber = firstNumber + secondNumber;
```

The error results from the fact that `thirdNumber` is a `short`. Although `firstNumber` and `secondNumber` are also `short` variables, the expression `firstNumber + secondNumber` results in an `int` value. The program can be corrected if `thirdNumber` is declared as an `int`, or if a cast operator is used in the assignment statement, as shown here:

```
thirdNumber = (short)(firstNumber + secondNumber);
```

## Other Mixed Mathematical Expressions

In situations where a mathematical expression has one or more values of the `double`, `float`, or `long` data types, Java strives to convert all of the operands in the expression to the same data type. Let's look at the specific rules that govern evaluation of these types of expressions.

1. If one of an operator's operands is a `double`, the value of the other operand will be converted to a `double`. The result of the expression will be a `double`. For example, in the following statement assume that `b` is a `double` and `c` is an `int`:

   ```
   a = b + c;
   ```

   The value in `c` will be converted to a `double` prior to the addition. The result of the addition will be a `double`, so the variable `a` must also be a `double`.
2. If one of an operator's operands is a `float`, the value of the other operand will be converted to a `float`. The result of the expression will be a `float`. For example, in the following statement assume that `x` is a `short` and `y` is a `float`:

   ```
   z = x * y;
   ```

   The value in `x` will be converted to a `float` prior to the multiplication. The result of the multiplication will be a `float`, so the variable `z` must also be either a `double` or a `float`.
3. If one of an operator's operands is a `long`, the value of the other operand will be converted to a `long`. The result of the expression will be a `long`. For example, in the following statement assume that `a` is a `long` and `b` is a `short`:

   ```
   c = a - b;
   ```

   The variable `b` will be converted to a `long` prior to the subtraction. The result of the subtraction will be a `long`, so the variable `c` must also be a `long`, `float`, or `double`.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.25 The following declaration appears in a program:

```
short totalPay, basePay = 500, bonus = 1000;
```

The following statement appears in the same program:

```
totalPay = basePay + bonus;
```

a) Will the statement compile properly or cause an error?
b) If the statement causes an error, why? How can you fix it?

2.26 The variable `a` is a `float` and the variable `b` is a `double`. Write a statement that will assign the value of `b` to `a` without causing an error when the program is compiled.

## 2.8 Creating Named Constants with `final`

**CONCEPT: The `final` key word can be used in a variable declaration to make the variable a named constant. Named constants are initialized with a value, and that value cannot change during the execution of the program.**

Assume that the following statement appears in a banking program that calculates data pertaining to loans:

```
amount = balance * 0.069;
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 8.2 percent? The programmer would have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using named constants. A *named constant* is a variable whose value is read only and cannot be changed during the program's execution. You can create such a variable in Java by using the `final` key word in the variable declaration. The word `final` is written just before the data type. Here is an example:

```
final double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word `final` appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

When you declare a variable as `final`, you must either initialize it, or assign it a value, before you can use it. For example, the following code declares and initializes a `final` variable named `PRICE` before it is used in an output operation:

```
final double PRICE = 19.99;
System.out.println("The price is " + PRICE);
```

The following code declares a `final` variable named `PRICE`, assigns it a value in a separate statement, and then uses it in an output operation:

```
final double PRICE;
PRICE = 19.99;
System.out.println("The price is " + PRICE);
```

The following code will cause an error when it is compiled, however, because you cannot use a `final` variable before you have given it a value:

```
final double PRICE;
System.out.println("The price is " + PRICE); // Error!
PRICE = 19.99;
```

Once you have given a `final` variable a value, no statement in the program can change its value. The following code will cause a compiler error:

```
final double PRICE;
PRICE = 19.99;
System.out.println("The price is " + PRICE);
PRICE = 12.99;  // Error! Cannot change PRICE.
```

An advantage of using named constants is that they make programs more self-documenting. The following statement:

```
amount = balance * 0.069;
```

can be changed to read

```
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know what is happening. It is evident that balance is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization value in the definition of the named constant is the only value that needs to be modified. If the rate increases to 8.2 percent, the declaration can be changed to the following:

```
final double INTEREST_RATE = 0.082;
```

The program is then ready to be recompiled. Every statement that uses `INTEREST_RATE` will use the new value.

### The `Math.PI` Named Constant

The `Math` class, which is part of the Java API, provides a predefined named constant, `Math.PI`. This constant is assigned the value 3.14159265358979323846, which is an approximation of the mathematical value pi. For example, look at the following statement:

```
area = Math.PI * radius * radius;
```

Assuming the `radius` variable holds the radius of a circle, this statement uses the `Math.PI` constant to calculate the area of the circle.

For more information about the `Math` class, see Appendix F, available on the book's companion Web site at www.pearsonhighered.com/gaddis.

## 2.9 The `String` Class

**CONCEPT: The `String` class allows you to create objects for holding strings. It also has various methods that allow you to work with strings.**

You have already encountered strings and examined programs that display them on the screen, but let's take a moment to make sure you understand what a string is. A string is a sequence of characters. It can be used to represent any type of data that contains text, such as names, addresses, warning messages, and so forth. String literals are enclosed in double-quotation marks, such as the following:

```
"Hello World"
"Joe Mahoney"
```

Although programs commonly encounter strings and must perform a variety of tasks with them, Java does not have a primitive data type for storing them in memory. Instead, the Java API provides a class for handling strings. You use this class to create objects that are capable of storing strings and performing operations on them. Before discussing this class, let's briefly discuss how classes and objects are related.

## Objects Are Created from Classes

Chapter 1 introduced you to objects as software entities that can contain attributes and methods. An object's attributes are data values that are stored in the object. An object's methods are procedures that perform operations on the object's attributes. Before an object can be created, however, it must be designed by a programmer. The programmer determines the attributes and methods that are necessary, and then creates a class that describes the object.

You have already seen classes used as containers for applications. A class can also be used to specify the attributes and methods that a particular type of object may have. Think of a class as a "blueprint" that objects may be created from. So a class is not an object, but a description of an object. When the program is running, it can use the class to create, in memory, as many objects as needed. Each object that is created from a class is called an *instance* of the class.

**TIP:** Don't worry if these concepts seem a little fuzzy to you. As you progress through this book, the concepts of classes and objects will be reinforced again and again.

## The `String` Class

The class that is provided by the Java API for handling strings is named `String`. The first step in using the `String` class is to declare a variable of the `String` class data type. Here is an example of a `String` variable declaration:

```
String name;
```

**TIP:** The `S` in `String` is written in an uppercase letter. By convention, the first character of a class name is always written in an uppercase letter.

This statement declares `name` as a `String` variable. Remember that `String` is a class, not a primitive data type. Let's briefly look at the difference between primitive type variables and class type variables.

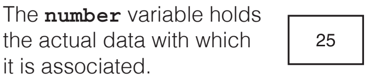
## Primitive Type Variables and Class Type Variables

A variable of any type can be associated with an item of data. *Primitive type variables* hold the actual data items with which they are associated. For example, assume that `number` is an `int` variable. The following statement stores the value 25 in the variable:

```
number = 25;
```

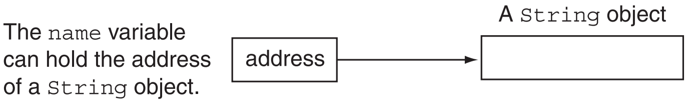
This is illustrated in Figure 2-7.

**Figure 2-7** A primitive type variable holds the data with which it is associated


The **number** variable holds
the actual data with which
it is associated.
25

A *class type variable* does not hold the actual data item that it is associated with, but holds the memory address of the data item it is associated with. If `name` is a `String` class variable, then name can hold the memory address of a `String` object. This is illustrated in Figure 2-8.

**Figure 2-8** A String class variable can hold the address of a String object


The name variable
can hold the address
of a String object.
address
A String object

When a class type variable holds the address of an object, it is said that the variable references the object. For this reason, class type variables are commonly known as *reference variables*.

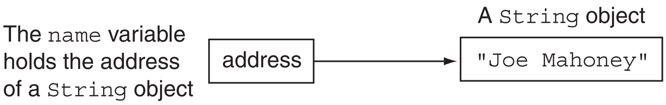
## Creating a `String` Object

Any time you write a string literal in your program, Java will create a `String` object in memory to hold it. You can create a `String` object in memory and store its address in a `String` variable with a simple assignment statement. Here is an example:

```
name = "Joe Mahoney";
```

Here, the string literal causes a `String` object to be created in memory with the value “Joe Mahoney” stored in it. Then the assignment operator stores the address of that object in the `name` variable. After this statement executes, it is said that the `name` variable references a `String` object. This is illustrated in Figure 2-9.

**Figure 2-9** The `name` variable holds the address of a `String` object


The name variable
holds the address
of a String object
address
A String object
"Joe Mahoney"

You can also use the = operator to initialize a `String` variable, as shown here:

```
String name = "Joe Mahoney";
```

This statement declares `name` as a `String` variable, creates a `String` object with the value “Joe Mahoney" stored in it, and assigns the object’s memory address to the `name` variable. Code Listing 2-20 shows `String` variables being declared, initialized, and then used in a `println` statement.

**Code Listing 2-20** (`StringDemo.java`)

```
// A simple program demonstrating String objects.

public class StringDemo
{
   public static void main(String[] args)
   {
      String greeting = "Good morning, ";
      String name = "Herman";

      System.out.println(greeting + name);
   }
}
```

**Program Output**

```
Good morning, Herman
```

Because the `String` type is a class instead of a primitive data type, it provides numerous methods for working with strings. For example, the `String` class has a method named `length` that returns the length of the string stored in an object. Assuming the `name` variable references a `String` object, the following statement stores the length of its string in the variable `stringSize` (assume that `stringSize` is an `int` variable):

```
stringSize = name.length();
```

This statement calls the `length` method of the object that `name` refers to. To *call* a method means to execute it. The general form of a method call is as follows:

```
referenceVariable.method(arguments . . . )
```

*`referenceVariable`* is the name of a variable that references an object, *`method`* is the name of a method, and *`arguments. . .`* is zero or more arguments that are passed to the method. If no arguments are passed to the method, as is the case with the `length` method, a set of empty parentheses must follow the name of the method.

The `String` class's `length` method *returns* an `int` value. This means that the method sends an `int` value back to the statement that called it. This value can be stored in a variable, displayed on the screen, or used in calculations. Code Listing 2-21 demonstrates the `length` method.

**Code Listing 2-21** (`StringLength.java`)

```
// This program demonstrates the String class's length method.

public class StringLength
```

```java
{
   public static void main(String[] args)
   {
      String name = "Herman";
      int stringSize;

      stringSize = name.length();
      System.out.println(name + " has " + stringSize +
                         " characters.");
   }
}
```

**Program Output**

```
Herman has 6 characters.
```

**NOTE:** The `String` class's `length` method returns the number of characters in the string, including spaces.

You will study the `String` class methods in detail in Chapter 9, but let's look at a few more examples now. In addition to `length`, Table 2-15 describes the `charAt`, `toLowerCase`, and `toUpperCase` methods.

**Table 2-15** A few `String` class methods

| Method | Description and Example |
|---|---|
| `charAt(`*index*`)` | The argument *index* is an `int` value and specifies a character position in the string. The first character is at position 0, the second character is at position 1, and so forth. The method returns the character at the specified position. The return value is of the type `char`.<br>**Example:**<br>`char letter;`<br>`String name = "Herman";`<br>`letter = name.charAt(3);`<br>After this code executes, the variable `letter` will hold the character 'm'. |
| `length()` | This method returns the number of characters in the string. The return value is of the type `int`.<br>**Example:**<br>`int stringSize;`<br>`String name = "Herman";`<br>`stringSize = name.length();` |

**Table 2-15** A few `String` class methods (*continued*)

| Method | Description and Example |
|---|---|
| | After this code executes, the `stringSize` variable will hold the value 6. |
| `toLowerCase()` | This method returns a new string that is the lowercase equivalent of the string contained in the calling object.<br>**Example:**<br>`String bigName = "HERMAN";`<br>`String littleName = bigName.toLowerCase();`<br>After this code executes, the object referenced by `littleName` will hold the string "herman". |
| `toUpperCase()` | This method returns a new string that is the uppercase equivalent of the string contained in the calling object.<br>**Example:**<br>`String littleName = "herman";`<br>`String bigName = littleName.toUpperCase();`<br>After this code executes, the object referenced by `bigName` will hold the string "HERMAN". |

The program in Code Listing 2-22 demonstrates these methods.

**Code Listing 2-22** (`StringMethods.java`)

```java
// This program demonstrates a few of the String methods.

public class StringMethods
 {
    public static void main(String[] args)
    {
      String message = "Java is Great Fun!";
      String upper = message.toUpperCase();
      String lower = message.toLowerCase();
      char letter = message.charAt(2);
      int stringSize = message.length();

      System.out.println(message);
      System.out.println(upper);
      System.out.println(lower);
      System.out.println(letter);
      System.out.println(stringSize);
   }
}
```

**Program Output**

```
Java is Great Fun!
JAVA IS GREAT FUN!
java is great fun!
v
18
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

2.27 Write a statement that declares a `String` variable named `city`. The variable should be initialized so it references an object with the string "San Francisco".

2.28 Assume that `stringLength` is an `int` variable. Write a statement that stores the length of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `stringLength`.

2.29 Assume that `oneChar` is a `char` variable. Write a statement that stores the first character in the string referenced by the `city` variable (declared in Checkpoint 2.27) in `oneChar`.

2.30 Assume that `upperCity` is a `String` reference variable. Write a statement that stores the uppercase equivalent of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `upperCity`.

2.31 Assume that `lowerCity` is a `String` reference variable. Write a statement that stores the lowercase equivalent of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `lowerCity`.

## 2.10 Scope

**CONCEPT: A variable's scope is the part of the program that has access to the variable.**

Every variable has a *scope*. The scope of a variable is the part of the program where the variable may be accessed by its name. A variable is visible only to statements inside the variable's scope. The rules that define a variable's scope are complex, and you are only introduced to the concept here. In other chapters of the book we revisit this topic and expand on it.

So far, you have only seen variables declared inside the `main` method. Variables that are declared inside a method are called *local variables*. Later you will learn about variables that are declared outside a method, but for now, let's focus on the use of local variables.

A local variable's scope begins at the variable's declaration and ends at the end of the method in which the variable is declared. The variable cannot be accessed by statements that are outside this region. This means that a local variable cannot be accessed by code that is outside the method, or inside the method but before the variable's declaration. The program in Code Listing 2-23 shows an example.

### Code Listing 2-23 (`Scope.java`)

```java
// This program can't find its variable.

public class Scope
{
   public static void main(String[] args)
   {
      System.out.println(value); // ERROR!
      int value = 100;
   }
}
```

The program does not compile because it attempts to send the contents of the variable `value` to `println` before the variable is declared. It is important to remember that the compiler reads your program from top to bottom. If it encounters a statement that uses a variable before the variable is declared, an error will result. To correct the program, the variable declaration must be written before any statement that uses it.

**NOTE:** If you compile this program, the compiler will display an error message such as "cannot resolve symbol." This means that the compiler has encountered a name for which it cannot determine a meaning.

Another rule that you must remember about local variables is that you cannot have two local variables with the same name in the same scope. For example, look at the following method.

```java
public static void main(String[] args)
{
   // Declare a variable named number and
   // display its value.
   int number = 7;
   System.out.println(number);
```

```
        // Declare another variable named number and
        // display its value.
        int number = 100;               // ERROR!!!
        System.out.println(number);     // ERROR!!!
    }
```

This method declares a variable named `number` and initializes it with the value 7. The variable's scope begins at the declaration statement and extends to the end of the method. Inside the variable's scope a statement appears that declares another variable named `number`. This statement will cause an error because you cannot have two local variables with the same name in the same scope.

## 2.11 Comments

**CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.**

Comments are short notes that are placed in different parts of a program, explaining how those parts of the program work. Comments are not intended for the compiler. They are intended for programmers to read, to help them understand the code. The compiler skips all of the comments that appear in a program.

As a beginning programmer, you might resist the idea of writing a lot of comments in your programs. After all, it's a lot more fun to write code that actually does something! However, it's crucial that you take the extra time to write comments. They will almost certainly save you time in the future when you have to modify or debug the program. Even large and complex programs can be made easy to read and understand if they are properly commented.

In Java there are three types of comments: single-line comments, multiline comments, and documentation comments. Let's briefly discuss each type.

### Single-Line Comments

You have already seen the first way to write comments in a Java program. You simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Code Listing 2-24 shows that comments may be placed liberally throughout a program.

**Code Listing 2-24** (`Comment1.java`)

```
// PROGRAM: Comment1.java
// Written by Herbert Dorfmann
// This program calculates company payroll

public class Comment1
```

```
{
   public static void main(String[] args)
   {
      double payRate;       // Holds the hourly pay rate
      double hours;         // Holds the hours worked
      int employeeNumber;   // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

In addition to telling who wrote the program and describing the purpose of variables, comments can also be used to explain complex procedures in your code.

### Multi-Line Comments

The second type of comment in Java is the multi-line comment. *Multi-line comments* start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Code Listing 2-25 illustrates how multi-line comments may be used.

**Code Listing 2-25** (`Comment2.java`)

```
/*
   PROGRAM: Comment2.java
   Written by Herbert Dorfmann
   This program calculates company payroll
*/

public class Comment2
{
   public static void main(String[] args)
   {
      double payRate;        // Holds the hourly pay rate
      double hours;          // Holds the hours worked
      int employeeNumber;    // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

Unlike a comment started with `//`, a multi-line comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to mark every line. Consequently, the multi-line comment is inconvenient for writing single-line comments because you must type both a beginning and an ending comment symbol.

Remember the following advice when using multi-line comments:

- Be careful not to reverse the beginning symbol with the ending symbol.
- Be sure not to forget the ending symbol.

Many programmers use asterisks or other characters to draw borders or boxes around their comments. This helps to visually separate the comments from surrounding code. These are called block comments. Table 2-16 shows four examples of block comments.

**Table 2-16** Block comments

```
/*
 *   This program demonstrates the
 *   way to write comments.
 */
```

```
//***********************************
//    This program demonstrates the *
//    way to write comments.        *
//***********************************
```

```
////////////////////////////////////
//   This program demonstrates the
//   way to write comments.
////////////////////////////////////
```

```
//-----------------------------------
//    This program demonstrates the
//    way to write comments.
//-----------------------------------
```

### Documentation Comments

The third type of comment is known as a documentation comment. *Documentation comments* can be read and processed by a program named `javadoc`, which comes with the JDK. The purpose of the `javadoc` program is to read Java source code files and generate attractively formatted HTML files that document the source code. If the source code files contain any documentation comments, the information in the comments becomes part of the HTML documentation. The HTML documentation files may be viewed in a Web browser.

Any comment that starts with `/**` and ends with `*/` is considered a documentation comment. Normally you write a documentation comment just before a class header, giving a brief description of the class. You also write a documentation comment just before each method header, giving a brief description of the method. For example, Code Listing 2-26 shows a program with documentation comments. This program has a documentation comment just before the class header, and just before the `main` method header.

**Code Listing 2-26** (`Comment3.java`)

```
/**
 This class creates a program that calculates company payroll.
*/

public class Comment3
{
   /**
      The main method is the program's starting point.
   */

   public static void main(String[] args)
   {
      double payRate;       // Holds the hourly pay rate
      double hours;         // Holds the hours worked
      int employeeNumber;   // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

You run the `javadoc` program from the operating system command prompt. Here is the general format of the `javadoc` command:

```
javadoc SourceFile.java
```

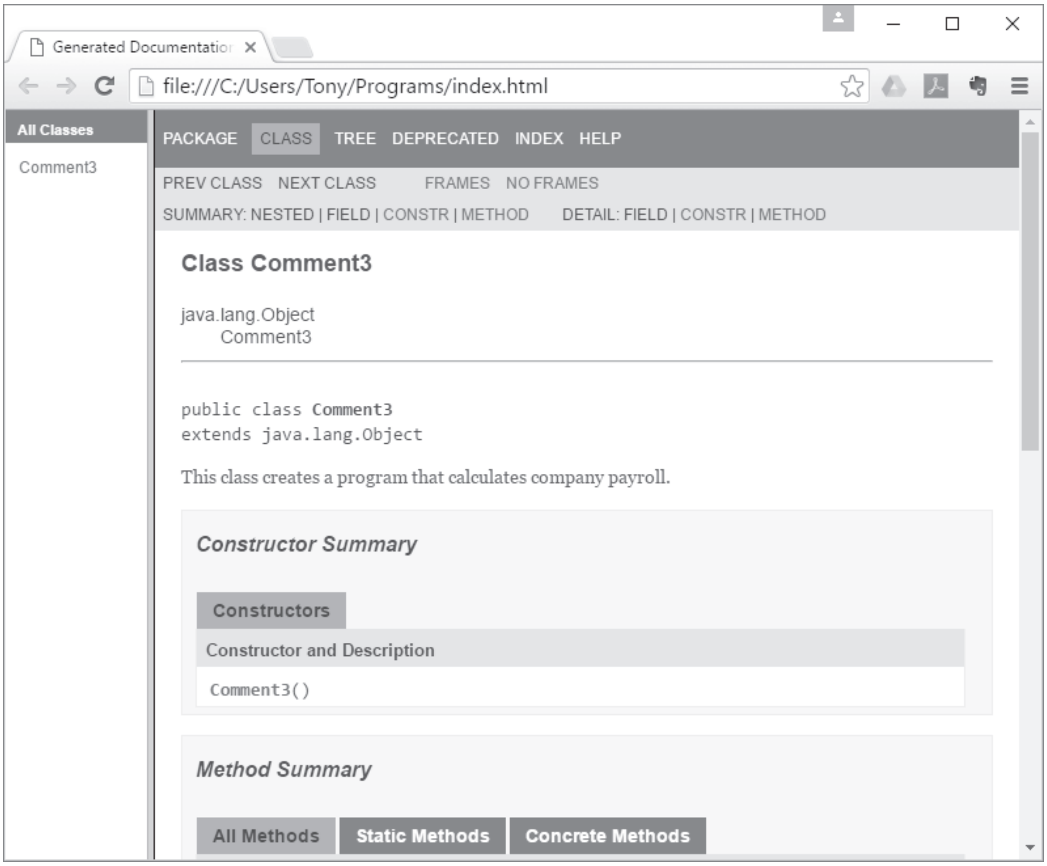
`SourceFile.java` is the name of a Java source code file, including the `.java` extension. The file will be read by `javadoc` and documentation will be produced for it. For example, the following command will produce documentation for the *Comment3.java* source code file, which is shown in Code Listing 2-26:

```
javadoc Comment3.java
```

After this command executes, several documentation files will be created in the same directory as the source code file. One of these files will be named *index.html*. Figure 2-10 shows the *index.html* file being viewed in a Web browser. Notice that the text that was written in the documentation comments appears in the file.

**TIP:** When you write a documentation comment for a method, the HTML documentation file that is produced by `javadoc` will have two sections for the method: a summary section and a detail section. The first sentence in the method's documentation comment is used as the summary of the method. Note that `javadoc` considers the end of the sentence as a period followed by a whitespace character. For this reason, when a method description contains more than one sentence, you should always end the first sentence with a period followed by a whitespace character. The method's detail section will contain all of the description that appears in the documentation comment.

**Figure 2-10** Documentation generated by `javadoc` (Google Inc.)


Generated Documentation
file:///C:/Users/Tony/Programs/index.html
All Classes
Comment3
PACKAGE
CLASS
TREE
DEPRECATED
INDEX
HELP
PREV CLASS
NEXT CLASS
FRAMES
NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class Comment3

java.lang.Object
Comment3

```
public class Comment3
extends java.lang.Object
```

This class creates a program that calculates company payroll.

Constructor Summary

| Constructors |
|---|
| Constructor and Description |
| `Comment3()` |

Method Summary

All Methods
Static Methods
Concrete Methods

If you look at the JDK documentation, which are HTML files that you view in a Web browser, you will see that they are formatted in the same way as the files generated by `javadoc`. A benefit of using `javadoc` to document your source code is that your documentation will have the same professional look and feel as the standard Java documentation.

From this point forward in the book, we will use documentation comments in the example source code. As we progress through various topics, you will see additional uses of documentation comments and the `javadoc` program.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.32 How do you write a single line comment? How do you write a multi-line comment? How do you write a documentation comment?

2.33 How are documentation comments different from other types of comments?

## 2.12 Programming Style

**CONCEPT: Programming style refers to the way a programmer uses spaces, indentations, blank lines, and punctuation characters to visually arrange a program's source code.**

In Chapter 1, you learned that syntax rules govern the way a language may be used. The syntax rules of Java dictate how and where to place key words, semicolons, commas, braces, and other elements of the language. The compiler checks for syntax errors, and if there are none, generates byte code.

When the compiler reads a program it processes it as one long stream of characters. The compiler doesn't care that each statement is on a separate line, or that spaces separate operators from operands. Humans, on the other hand, find it difficult to read programs that aren't written in a visually pleasing manner. Consider Code Listing 2-27 for example.

**Code Listing 2-27** (`Compact.java`)

```
public class Compact {public static void main(String [] args){int
shares=220; double averagePrice=14.67; System.out.println(
"There were "+shares+" shares sold at $"+averagePrice+
" per share.");}}
```

**Program Output**

```
There were 220 shares sold at $14.67 per share.
```

Although the program is syntactically correct (it doesn't violate any rules of Java), it is very difficult to read. The same program is shown in Code Listing 2-28, written in a more understandable style.

**Code Listing 2-28** (`Readable.java`)

```
/**
  This example is much more readable than Compact.java.
*/

public class Readable
{
   public static void main(String[] args)
   {
      int shares = 220;
      double averagePrice = 14.67;

      System.out.println("There were " + shares +
                         " shares sold at $" +
                         averagePrice + " per share.");
   }
}
```

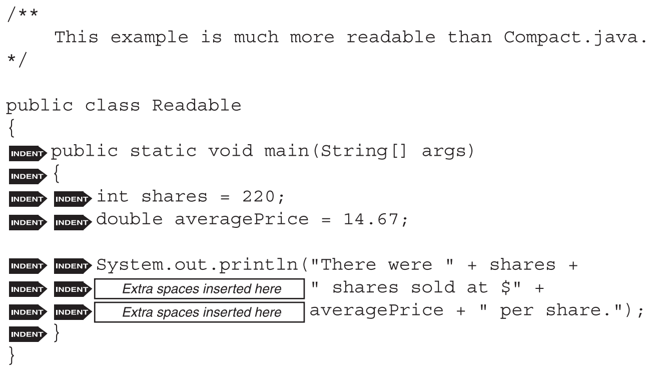
**Program Output**

```
There were 220 shares sold at $14.67 per share.
```

The term *programming style* usually refers to the way source code is visually arranged. It includes techniques for consistently putting spaces and indentations in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Code Listing 2-28 that inside the class's braces each line is indented, and inside the `main` method's braces each line is indented again. It is a common programming style to indent all the lines inside a set of braces, as shown in Figure 2-11.

**Figure 2-11** Indentation



```
/**
    This example is much more readable than Compact.java.
*/

public class Readable
{
    public static void main(String[] args)
    {
        int shares = 220;
        double averagePrice = 14.67;

        System.out.println("There were " + shares +
                           " shares sold at $" +
                           averagePrice + " per share.");
    }
}
```

Another aspect of programming style is how to handle statements that are too long to fit on one line. Notice that the `println` statement is spread out over three lines. Extra spaces are inserted at the beginning of the statement's second and third lines, which indicate that they are continuations.

When declaring multiple variables of the same type with a single statement, it is a common practice to write each variable name on a separate line with a comment explaining the variable's purpose. Here is an example:

```
int fahrenheit,    // To hold the Fahrenheit temperature
    celsius,       // To hold the Celsius temperature
    kelvin;        // To hold the Kelvin temperature
```

You may have noticed in the example programs that a blank line is inserted between the variable declarations and the statements that follow them. This is intended to separate the declarations visually from the executable statements.

There are many other issues related to programming style. They will be presented throughout the book.

## 2.13 Reading Keyboard Input

**CONCEPT: Objects of the `Scanner` class can be used to read input from the keyboard.**

Previously we discussed the `System.out` object, and how it refers to the standard output device. The Java API has another object, `System.in`, which refers to the standard input device. The *standard input device* is normally the keyboard. You can use the `System.in` object to read keystrokes that have been typed at the keyboard. However, using `System.in` is not as simple and straightforward as using `System.out` because the `System.in` object reads input only as `byte` values. This isn't very useful because programs normally require values of other data types as input. To work around this, you can use the `System.in` object in conjunction with an object of the `Scanner` class. The `Scanner` class is designed to read input from a source (such as `System.in`), and it, provides methods that you can use to retrieve the input formatted as primitive values or strings.

First, you create a `Scanner` object and connect it to the `System.in` object. Here is an example of a statement that does just that:

```
Scanner keyboard = new Scanner(System.in);
```

Let's dissect the statement into two parts. The first part of the statement,

```
Scanner keyboard
```

declares a variable named `keyboard`. The data type of the variable is `Scanner`. Because `Scanner` is a class, the `keyboard` variable is a class type variable. Recall from our discussion on `String` objects that a class type variable holds the memory address of an object. Therefore, the `keyboard` variable will be used to hold the address of a `Scanner` object. The second part of the statement is as follows:

```
= new Scanner(System.in);
```

The first thing we see in this part of the statement is the assignment operator (`=`). The assignment operator will assign something to the `keyboard` variable. After the assignment operator we see the word `new`, which is a Java key word. The purpose of the `new` key word is to create an object in memory. The type of object that will be created is listed next. In this case, we see `Scanner(System.in)` listed after the `new` key word. This specifies that a `Scanner` object should be created, and it should be connected to the `System.in` object. The memory address of the object is assigned (by the `=` operator) to the variable `keyboard`. After the statement executes, the `keyboard` variable will reference the `Scanner` object that was created in memory.

Figure 2-12 points out the purpose of each part of this statement. Figure 2-13 illustrates how the `keyboard` variable references an object of the `Scanner` class.

**Figure 2-12** The parts of the statement

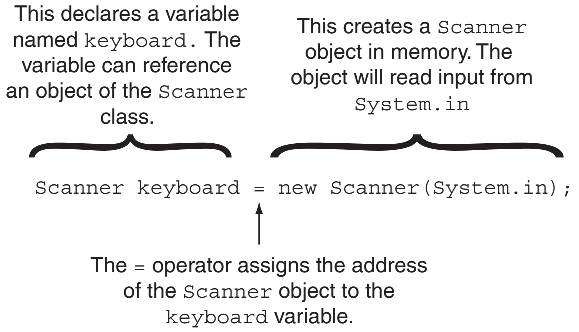
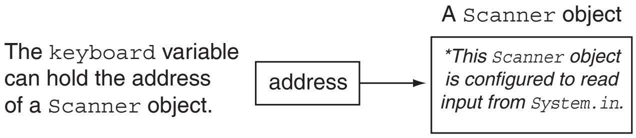

This declares a variable named keyboard. The variable can reference an object of the Scanner class.
This creates a Scanner object in memory. The object will read input from System.in
Scanner keyboard = new Scanner(System.in);
The = operator assigns the address of the Scanner object to the keyboard variable.

**Figure 2-13** The keyboard variable references a Scanner object


A Scanner object
The keyboard variable can hold the address of a Scanner object.
address
*This Scanner object is configured to read input from System.in.

**NOTE:** In the preceding code, we chose `keyboard` as the variable name. There is nothing special about the name `keyboard`. We simply chose that name because we will use the variable to read input from the keyboard.

The `Scanner` class has methods for reading strings, `byte`s, integers, `long` integers, `short` integers, `float`s, and `double`s. For example, the following code uses an object of the `Scanner` class to read an `int` value from the keyboard and assign the value to the `number` variable.

```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer value: ");
number = keyboard.nextInt();
```

The last statement shown here calls the `Scanner` class's `nextInt` method. The `nextInt` method formats an input value as an `int`, and then returns that value. Therefore, this statement formats the input that was entered at the keyboard as an `int`, and then returns it. The value is assigned to the `number` variable.

Table 2-17 lists several of the `Scanner` class's methods and describes their use.

**Table 2-17** Some of the `Scanner` class methods

| Method | Example and Description |
|---|---|
| `nextByte` | **Example Usage:** |

```
byte x;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a byte value: ");
x = keyboard.nextByte();
```

**Description:** Returns input as a `byte`.

| Method | Example and Description |
|---|---|
| `nextDouble` | **Example Usage:** |

```
double number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a double value: ");
number = keyboard.nextDouble();
```

**Description:** Returns input as a `double`.

| Method | Example and Description |
|---|---|
| `nextFloat` | **Example Usage:** |

```
float number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a float value: ");
number = keyboard.nextFloat();
```

**Description:** Returns input as a `float`.

| Method | Example and Description |
|---|---|
| `nextInt` | **Example Usage:** |

```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer value: ");
number = keyboard.nextInt();
```

**Description:** Returns input as an `int`.

| Method | Example and Description |
|---|---|
| `nextLine` | **Example Usage:** |

```
String name;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your name: ");
name = keyboard.nextLine();
```

**Description:** Returns input as a `String`.

| Method | Example and Description |
|---|---|
| `nextLong` | **Example Usage:** |

```
long number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a long value: ");
number = keyboard.nextLong();
```

**Description:** Returns input as a `long`.

| Method | Example and Description |
|---|---|
| `nextShort` | **Example Usage:** |

```
short number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a short value: ");
number = keyboard.nextShort();
```

**Description:** Returns input as a `short`.

### Using the `import` Statement

There is one last detail about the `Scanner` class that you must know before you will be ready to use it. The `Scanner` class is not automatically available to your Java programs. Any program that uses the `Scanner` class should have the following statement near the beginning of the file, before any class definition:

```
import java.util.Scanner;
```

This statement tells the Java compiler where in the Java library to find the `Scanner` class, and makes it available to your program.

Code Listing 2-29 shows the `Scanner` class being used to read a `String`, an `int`, and a `double`.

**Code Listing 2-29** (`Payroll.java`)

```
import java.util.Scanner; // Needed for the Scanner class

/**
  This program demonstrates the Scanner class.
*/

public class Payroll
{
   public static void main(String[] args)
   {
      String name;          // To hold a name
      int hours;            // Hours worked
      double payRate;       // Hourly pay rate
      double grossPay;      // Gross pay

      // Create a Scanner object to read input.
      Scanner keyboard = new Scanner(System.in);

      // Get the user's name.
      System.out.print("What is your name? ");
      name = keyboard.nextLine();

      // Get the number of hours worked this week.
      System.out.print("How many hours did you work this week? ");
      hours = keyboard.nextInt();

      // Get the user's hourly pay rate.
      System.out.print("What is your hourly pay rate? ");
      payRate = keyboard.nextDouble();

      // Calculate the gross pay.
      grossPay = hours * payRate;

```

```
        // Display the resulting information.
        System.out.println("Hello, " + name);
        System.out.println("Your gross pay is $" + grossPay);
    }
}
```

**Program Output with Example Input Shown in Bold**

```
What is your name? Joe Mahoney [Enter]
How many hours did you work this week? 40 [Enter]
What is your hourly pay rate? 20 [Enter]
Hello, Joe Mahoney
Your gross pay is $800.0
```

**NOTE:** Notice that each `Scanner` class method that we used waits for the user to press the Enter key before it returns a value. When the Enter key is pressed, the cursor automatically moves to the next line for subsequent output operations.

## Reading a Character

Sometimes you will want to read a single character from the keyboard. For example, your program might ask the user a yes/no question, and specify that he or she type Y for yes or N for no. The `Scanner` class does not have a method for reading a single character, however. The approach that we will use in this book for reading a character is to use the `Scanner` class's `nextLine` method to read a string from the keyboard, and then use the `String` class's `charAt` method to extract the first character of the string. This will be the character that the user entered at the keyboard. Here is an example:

```
String input;  // To hold a line of input
char answer;   // To hold a single character

// Create a Scanner object for keyboard input.
Scanner keyboard = new Scanner(System.in);

// Ask the user a question.
System.out.print("Are you having fun? (Y=yes, N=no) ");
input = keyboard.nextLine(); // Get a line of input.
answer = input.charAt(0);    // Get the first character.
```

The `input` variable references a `String` object. The last statement in this code calls the `String` class's `charAt` method to retrieve the character at position 0, which is the first character in the string. After this statement executes, the `answer` variable will hold the character that the user typed at the keyboard.

## Mixing Calls to `nextLine` with Calls to Other `Scanner` Methods

When you call one of the `Scanner` class's methods to read a primitive value, such as `nextInt` or `nextDouble`, and then call the `nextLine` method to read a string, an annoying and hard-to-find problem can occur. For example, look at the program in Code Listing 2-30.

**Code Listing 2-30** `(InputProblem.java)`

```
 1  import java.util.Scanner; // Needed for the Scanner class
 2
 3  /**
 4     This program has a problem reading input.
 5  */
 6
 7  public class InputProblem
 8  {
 9     public static void main(String[] args)
10     {
11        String name;    // To hold the user's name
12        int age;        // To hold the user's age
13        double income;  // To hold the user's income
14
15        // Create a Scanner object to read input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Get the user's age.
19        System.out.print("What is your age? ");
20        age = keyboard.nextInt();
21
22        // Get the user's income
23        System.out.print("What is your annual income? ");
24        income = keyboard.nextDouble();
25
26        // Get the user's name.
27        System.out.print("What is your name? ");
28        name = keyboard.nextLine();
29
30        // Display the information back to the user.
31        System.out.println("Hello, " + name + ". Your age is " +
32                           age + " and your income is $" +
33                           income);
34     }
35  }
```

**Program Output with Example Input Shown in Bold**

```
What is your age? 24 [Enter]
What is your annual income? 50000.00 [Enter]
What is your name? Hello, . Your age is 24 and your income is $50000.0
```

Notice in the example output that the program first allows the user to enter his or her age. The statement in line 20 reads an `int` from the keyboard and stores the value in the `age` variable. Next, the user enters his or her income. The statement in line 24 reads a `double` from the keyboard and stores the value in the `income` variable. Then the user is asked to enter his or her name, but it appears that the statement in line 28 is skipped. The name is never read from the keyboard. This happens because of a slight difference in behavior between the `nextLine` method and the other `Scanner` class methods.

When the user types keystrokes at the keyboard, those keystrokes are stored in an area of memory that is sometimes called the *keyboard buffer*. Pressing the Enter ↵ key causes a newline character to be stored in the keyboard buffer. In the example running of the program in Code Listing 2-30, the user was asked to enter his or her age, and the statement in line 20 called the `nextInt` method to read an integer from the keyboard buffer. Notice that the user typed 24 and then pressed the Enter ↵ key. The `nextInt` method read the value 24 from the keyboard buffer, and then stopped when it encountered the newline character. So the value 24 was read from the keyboard buffer, but the newline character was not read. The newline character remained in the keyboard buffer.

Next, the user was asked to enter his or her annual income. The user typed 50000.00 and then pressed the Enter ↵ key. When the `nextDouble` method in line 24 executed, it first encountered the newline character that was left behind by the `nextInt` method. This does not cause a problem because the `nextDouble` method is designed to skip any leading newline characters it encounters. It skips over the initial newline, reads the value 50000.00 from the keyboard buffer, and stops reading when it encounters the next newline character. This newline character is then left in the keyboard buffer.

Next, the user is asked to enter his or her name. In line 28 the `nextLine` method is called. The `nextLine` method, however, is not designed to skip over an initial newline character. If a newline character is the first character that the `nextLine` method encounters, then nothing will be read. Because the `nextDouble` method, back in line 24, left a newline character in the keyboard buffer, the `nextLine` method will not read any input. Instead, it will immediately terminate and the user will not be given a chance to enter his or her name.

Although the details of this problem might seem confusing, the solution is easy. The program in Code Listing 2-31 is a modification of Code Listing 2-30, with the input problem fixed.

**Code Listing 2-31** (`CorrectedInputProblem.java`)

```
import java.util.Scanner;  // Needed for the Scanner class

/**
   This program correctly reads numeric and string input.
*/

public class CorrectedInputProblem
{
   public static void main(String[] args)
   {
      String name;     // To hold the user's name
12         int age;        // To hold the user's age
13         double income;  // To hold the user's income
14 
15         // Create a Scanner object to read input.
16         Scanner keyboard = new Scanner(System.in);
17 
18         // Get the user's age.
19         System.out.print("What is your age? ");
20         age = keyboard.nextInt();
21 
22         // Get the user's income
23         System.out.print("What is your annual income? ");
24         income = keyboard.nextDouble();
25 
26         // Consume the remaining newline.
27         keyboard.nextLine();
28 
29         // Get the user's name.
30         System.out.print("What is your name? ");
31         name = keyboard.nextLine();
32 
33         // Display the information back to the user.
34         System.out.println("Hello, " + name + ". Your age is " +
35                            age + " and your income is $" +
36                            income);
37      }
38   }
```

**Program Output with Example Input Shown in Bold**

```
What is your age? 24 [Enter]
What is your annual income? 50000.00 [Enter]
What is your name? Mary Simpson [Enter]
Hello, Mary Simpson. Your age is 24 and your income is $50000.0
```

Notice that after the user's income is read by the `nextDouble` method in line 24, the `nextLine` method is called in line 27. The purpose of this call is to consume, or remove, the newline character that remains in the keyboard buffer. Then, in line 31, the `nextLine` method is called again. This time it correctly reads the user's name.

**NOTE:** Notice that in line 27, where we consume the remaining newline character, we do not assign the method's return value to any variable. This is because we are simply calling the method to remove the newline character, and we do not need to keep the method's return value.

## 2.14 Dialog Boxes

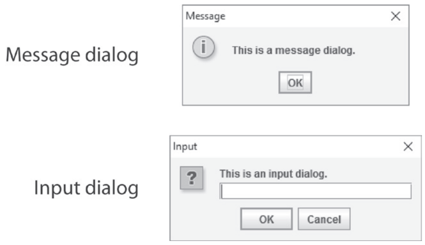
**CONCEPT: The `JOptionPane` class allows you to quickly display a dialog box, which is a small graphical window displaying a message or requesting input.**

A *dialog box* is a small graphical window that displays a message to the user or requests input. You can quickly display dialog boxes with the `JOptionPane` class. In this section we will discuss the following types of dialog boxes and how you can display them using `JOptionPane`:

| | |
|---|---|
| • Message Dialog | A dialog box that displays a message; an OK button is also displayed |
| • Input Dialog | A dialog box that prompts the user for input and provides a text field where input is typed; an OK button and a Cancel button are also displayed |

Figure 2-14 shows an example of each type of dialog box.

**Figure 2-14** A message dialog and an input dialog (Oracle Corporate Counsel)



This is a message dialog.

This is an input dialog.

The `JOptionPane` class is not automatically available to your Java programs. Any program that uses the `JOptionPane` class must have the following statement near the beginning of the file:

```
import javax.swing.JOptionPane;
```

This statement tells the compiler where to find the `JOptionPane` class and makes it available to your program.

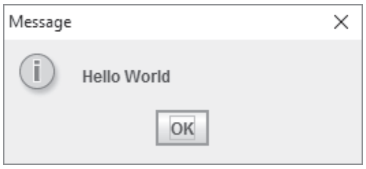
### Displaying Message Dialogs

The `showMessageDialog` method is used to display a message dialog. Here is a statement that calls the method:

```
JOptionPane.showMessageDialog(null, "Hello World");
```

The first argument is only important in programs that display other graphical windows. In our examples, we will always pass the key word `null` as the first argument. This causes the dialog box to be displayed in the center of the screen. The second argument is the message that we wish to display in the dialog box. This code will cause the dialog box in Figure 2-15 to appear. When the user clicks the OK button, the dialog box will close.

**Figure 2-15** Message dialog (Oracle Corporate Counsel)



Hello World

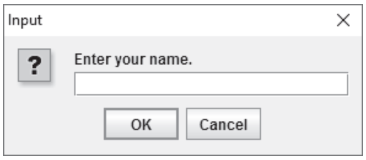
## Displaying Input Dialogs

An input dialog is a quick and simple way to ask the user to enter data. You use the `JOptionPane` class's `showInputDialog` method to display an input dialog. The following code calls the method:

```
String name;
name = JOptionPane.showInputDialog("Enter your name.");
```

The argument passed to the method is a message to display in the dialog box. This statement will cause the dialog box shown in Figure 2-16 to be displayed in the center of the screen. If the user clicks the OK button, `name` will reference the string value entered by the user into the text field. If the user clicks the Cancel button, `name` will reference the special value `null`.

**Figure 2-16** Input dialog (Oracle Corporate Counsel)



Enter your name.

## An Example Program

The program in Code Listing 2-32 demonstrates how to use both types of dialog boxes. This program uses input dialogs to ask the user to enter his or her first, middle, and last names, and then displays a greeting with a message dialog. When this program executes, the dialog boxes shown in Figure 2-17 will be displayed, one at a time.

**Code Listing 2-32** (`NamesDialog.java`)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates using dialogs
   with JOptionPane.
*/

public class NamesDialog
{
   public static void main(String[] args)
   {
      String firstName, lastName;

      // Get the user's first name.
      firstName =
         JOptionPane.showInputDialog("What is your first name?");

      // Get the user's last name.
      lastName =
         JOptionPane.showInputDialog("What is your last name?");

      // Display a greeting
      JOptionPane.showMessageDialog(null, "Hello " + firstName +
                                    " " + lastName);
      System.exit(0);
   }
}
```

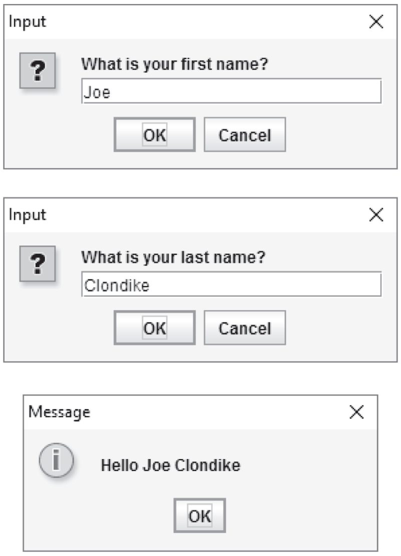
Notice the last statement in the `main` method:

```
System.exit(0);
```

This statement causes the program to end, and is necessary when you use the `JOptionPane` class in a console program. When a console program uses `JOptionPane` to display a dialog box, the program does not automatically stop executing when the end of the `main` method is reached. This is because the `JOptionPane` class causes an additional task to run in the JVM. If the `System.exit` method is not called, this task, also known as a *thread*, will continue to execute, even after the end of the `main` method has been reached.

**WARNING!** Use `System.exit` only when absolutely necessary. In a large program, `System.exit` can cause problems because it unconditionally shuts down the JVM, bypassing the program's normal, logical flow. In this book we use `System.exit` only because it is necessary to end a console program that uses `JOptionPane`.

**Figure 2-17** Dialog boxes displayed by the `NamesDialog` program (Oracle Corporate Counsel)



What is your first name?
Joe

What is your last name?
Clondike

Hello Joe Clondike

The first dialog box appears as shown here. The user types Joe and clicks OK.

The second dialog box appears, as shown here. In this example the user types Clondike and clicks OK.

The third dialog box appears, as shown here, displaying a greeting.

The `System.exit` method requires an integer argument. This argument is an exit code that is passed back to the operating system. Although this code is usually ignored, it can be used outside the program to indicate whether the program ended successfully or as the result of a failure. The value 0 traditionally indicates that the program ended successfully.

## Converting String Input to Numbers

Unlike the `Scanner` class, the `JOptionPane` class does not have different methods for reading values of different data types as input. The `showInputDialog` method always returns the user's input as a `String`, even if the user enters numeric data. For example, if the user enters the number 72 into an input dialog, the `showInputDialog` method will return the string `"72"`. This can be a problem if you wish to use the user's input in a math operation because, as you know, you cannot perform math on strings. In such a case, you must convert the input to a numeric value. To convert a string value to a numeric value, you use one of the methods listed in Table 2-18.

**Table 2-18** Methods for converting strings to numbers

| Method | Use This Method To . . . | Example Code |
|---|---|---|
| `Byte.parseByte` | Convert a string to a `byte`. | `byte num;`<br>`num = Byte.parseByte(str);` |
| `Double.parseDouble` | Convert a string to a `double`. | `double num;`<br>`num = Double.parseDouble(str);` |
| `Float.parseFloat` | Convert a string to a `float`. | `float num;`<br>`num = Float.parseFloat(str);` |
| `Integer.parseInt` | Convert a string to an `int`. | `int num;`<br>`num = Integer.parseInt(str);` |
| `Long.parseLong` | Convert a string to a `long`. | `long num;`<br>`num = Long.parseLong(str);` |
| `Short.parseShort` | Convert a string to a `short`. | `short num;`<br>`num = Short.parseShort(str);` |

**NOTE:** The methods in Table 2-18 are part of Java's wrapper classes, which you will learn more about in Chapter 9.

Here is an example of how you would use the `Integer.parseInt` method to convert the value returned from the `JOptionPane.showInputDialog` method to an `int`:

```
int number;
String str;
str = JOptionPane.showInputDialog("Enter a number.");
number = Integer.parseInt(str);
```

After this code executes, the `number` variable will hold the value entered by the user, converted to an `int`. Here is an example of how you would use the `Double.parseDouble` method to convert the user's input to a `double`:

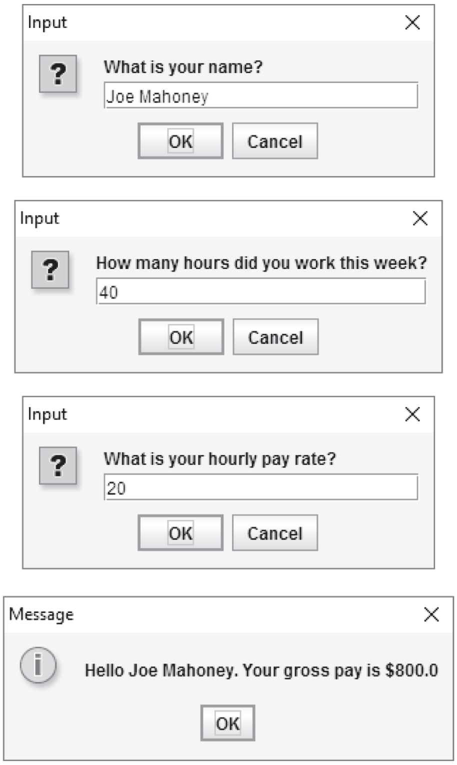
```
double price;
String str;
str = JOptionPane.showInputDialog("Enter the retail price.");
price = Double.parseDouble(str);
```

After this code executes, the `price` variable will hold the value entered by the user, converted to a `double`. Code Listing 2-33 shows a complete program. This is a modification of the *Payroll.java* program in Code Listing 2-29. When this program executes, the dialog boxes shown in Figure 2-18 will be displayed, one at a time.

**Code Listing 2-33** (`PayrollDialog.java`)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates using dialogs with
   JOptionPane.
*/

public class PayrollDialog
{
   public static void main(String[] args)
   {
      String inputString;         // For reading input
      String name;                // The user's name
      int hours;                  // The number of hours worked
      double payRate;             // The user's hourly pay rate
      double grossPay;            // The user's gross pay

      // Get the user's name.
      name = JOptionPane.showInputDialog("What is " +
                                          "your name? ");

      // Get the hours worked.
      inputString =
         JOptionPane.showInputDialog("How many hours " +
                                     "did you work this week? ");

      // Convert the input to an int.
      hours = Integer.parseInt(inputString);

      // Get the hourly pay rate.
      inputString =
         JOptionPane.showInputDialog("What is your " +
                                     "hourly pay rate? ");

      // Convert the input to a double.
      payRate = Double.parseDouble(inputString);

      // Calculate the gross pay.
      grossPay = hours * payRate;

      // Display the results.
      JOptionPane.showMessageDialog(null, "Hello " +
                      name + ". Your gross pay is $" +
                      grossPay);

      // End the program.
      System.exit(0);
   }
}
```

**Figure 2-18** Dialog boxes displayed by `PayrollDialog.java` (Oracle Corporate Counsel)



The first dialog box appears as shown here. The user enters his or her name and then clicks OK.

What is your name?

Joe Mahoney

The second dialog box appears, as shown here. The user enters the number of hours workded and then clicks OK.

How many hours did you work this week?

40

The third dialog box appears, as shown here. The user enters his or her hourly pay rate and then clicks OK.

What is your hourly pay rate?

20

The fourth dialog box appears, as shown here.

Hello Joe Mahoney. Your gross pay is $800.0

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

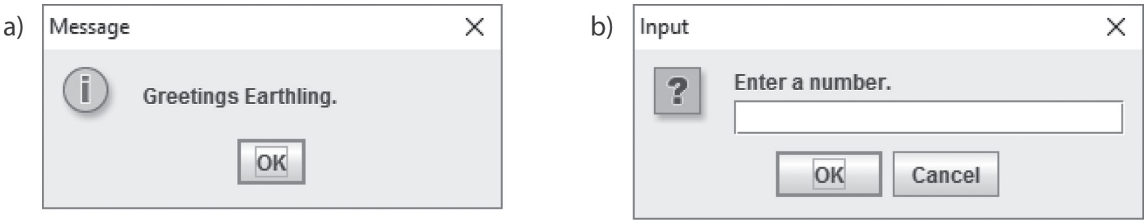
2.34 What is the purpose of the following types of dialog boxes?

Message dialog

Input dialog

2.35 Write code that will display each of the dialog boxes shown in Figure 2-19.

**Figure 2-19** Dialog boxes (Oracle Corporate Counsel)



Greetings Earthling.

Enter a number.

2.36 Write code that displays an input dialog asking the user to enter his or her age. Convert the input value to an `int` and store it in an `int` variable named `age`.

2.37 What `import` statement do you write in a program that uses the `JOptionPane` class?

## 2.15 Common Errors to Avoid

- **Mismatched braces, quotation marks, or parentheses.** In this chapter you saw that the statements making up a class definition are enclosed in a set of braces. Also, you saw that the statements in a method are also enclosed in a set of braces. For every opening brace, there must be a closing brace in the proper location. The same is true of double-quotation marks that enclose string literals and single-quotation marks that enclose character literals. Also, in a statement that uses parentheses, such as a mathematical expression, you must have a closing parenthesis for every opening parenthesis.
- **Misspelling key words.** Java will not recognize a key word that has been misspelled.
- **Using capital letters in key words.** Remember that Java is a case-sensitive language, and all key words are written in lowercase. Using an uppercase letter in a key word is the same as misspelling the key word.
- **Using a key word as a variable name.** The key words are reserved for special uses; they cannot be used for any other purpose.
- **Using inconsistent spelling of variable names.** Each time you use a variable name, it must be spelled exactly as it appears in its declaration statement.
- **Using inconsistent case of letters in variable names.** Because Java is a case-sensitive language, it distinguishes between uppercase and lowercase letters. Java will not recognize a variable name that is not written exactly as it appears in its declaration statement.
- **Inserting a space in a variable name.** Spaces are not allowed in variable names. Instead of using a two-word name such as `gross pay`, use one word, such as `grossPay`.
- **Forgetting the semicolon at the end of a statement.** A semicolon appears at the end of each complete statement in Java.
- **Assigning a `double` literal to a `float` variable.** Java is a strongly typed language, which means that it only allows you to store values of compatible data types in variables. All floating-point literals are treated as `double`s, and a `double` value is not compatible with a `float` variable. A floating-point literal must end with the letter `f` or `F` in order to be stored in a `float` variable.
- **Using commas or other currency symbols in numeric literals.** Numeric literals cannot contain commas or currency symbols, such as the dollar sign.

- **Unintentionally performing integer division.** When both operands of a division statement are integers, the statement will result in an integer. If there is a remainder, it will be discarded.
- **Forgetting to group parts of a mathematical expression.** If you use more than one operator in a mathematical expression, the expression will be evaluated according to the order of operations. If you wish to change the order in which the operators are used, you must use parentheses to group part of the expression.
- **Inserting a space in a combined assignment operator.** A space cannot appear between the two operators that make a combined assignment operator.
- **Using a variable to receive the result of a calculation when the variable's data type is incompatible with the data type of the result.** A variable that receives the result of a calculation must be of a data type that is compatible with the data type of the result.
- **Incorrectly terminating a multi-line comment or a documentation comment.** Multi-line comments and documentation comments are terminated by the `*/` characters. Forgetting to place these characters at a comment's desired ending point, or accidentally switching the `*` and the `/`, will cause the comment not to have an ending point.
- **Forgetting to use the correct `import` statement in a program that uses the `Scanner` class or the `JOptionPane` class.** In order for the `Scanner` class to be available to your program, you must have the import `java.util.Scanner;` statement near the top of your program file. In order for the `JOptionPane` class to be available to your program, you must have the import `javax.swing.JOptionPane;` statement near the top of the program file.
- **When using an input dialog to read numeric input, not converting the `showInputDialog` method's return value to a number.** The `showInputDialog` method always returns the user's input as a string. If the user enters a numeric value, it must be converted to a number before it can be used in a math statement.

## Review Questions and Exercises

### Multiple Choice and True/False

1. Every complete statement ends with a __________.
   a. period
   b. parenthesis
   c. semicolon
   d. ending brace
2. The following data

```
72
'A'
"Hello World"
2.8712
```

   are all examples of __________.
   a. variables
   b. literals
   c. strings
   d. none of these

3. A group of statements, such as the contents of a class or a method, are enclosed in __________.
   a. braces `{}`
   b. parentheses `()`
   c. brackets `[]`
   d. any of these will do
4. Which of the following are *not* valid assignment statements? (Indicate all that apply.)
   a. `total = 9;`
   b. `72 = amount;`
   c. `profit = 129`
   d. `letter = 'W';`
5. Which of the following are not valid `println` statements? (Indicate all that apply.)
   a. `System.out.println + "Hello World";`
   b. `System.out.println("Have a nice day");`
   c. `out.System.println(value);`
   d. `println.out(Programming is great fun);`
6. The negation operator is __________.
   a. unary
   b. binary
   c. ternary
   d. none of these
7. This key word is used to declare a named constant.
   a. `constant`
   b. `namedConstant`
   c. `final`
   d. `concrete`
8. These characters mark the beginning of a multi-line comment.
   a. `//`
   b. `/*`
   c. `*/`
   d. `/**`
9. These characters mark the beginning of a single-line comment.
   a. `//`
   b. `/*`
   c. `*/`
   d. `/**`
10. These characters mark the beginning of a documentation comment.
    a. `//`
    b. `/*`
    c. `*/`
    d. `/**`
11. Which `Scanner` class method would you use to read a string as input?
    a. `nextString`
    b. `nextLine`
    c. `readString`
    d. `getLine`

12. Which `Scanner` class method would you use to read a `double` as input?
    a. `nextDouble`
    b. `getDouble`
    c. `readDouble`
    d. None of these; you cannot read a `double` with the `Scanner` class
13. You can use this class to display dialog boxes.
    a. `JOptionPane`
    b. `BufferedReader`
    c. `InputStreamReader`
    d. `DialogBox`
14. When Java converts a lower-ranked value to a higher-ranked type, it is called a(n) __________.
    a. 4-bit conversion
    b. escalating conversion
    c. widening conversion
    d. narrowing conversion
15. This type of operator lets you manually convert a value, even if it means that a narrowing conversion will take place.
    a. cast
    b. binary
    c. uploading
    d. dot
16. **True or False:** A left brace in a Java program is always followed by a right brace later in the program.
17. **True or False:** A variable must be declared before it can be used.
18. **True or False:** Variable names may begin with a number.
19. **True or False:** You cannot change the value of a variable whose declaration uses the `final` key word.
20. **True or False:** Comments that begin with `//` can be processed by `javadoc`.
21. **True or False:** If one of an operator's operands is a `double`, and the other operand is an `int`, Java will automatically convert the value of the `double` to an `int`.

## Predict the Output

What will the following code segments print on the screen?

1.
```
int freeze = 32, boil = 212;
freeze = 0;
boil = 100;
System.out.println(freeze + "\n"+ boil + "\n");
```

2.
```
int x = 0, y = 2;
x = y * 4;
System.out.println(x + "\n" + y + "\n");
```

3.
```
System.out.print("I am the incredible");
System.out.print("computing\nmachine");
System.out.print("\nand I will\namaze\n)";
System.out.println("you.");
```

4.
```
System.out.print("Be careful\n)";
System.out.print("This might/n be a trick ");
System.out.println("question.");
```

5.
```
int a, x = 23;
a = x % 2;
System.out.println(x + "\n" + a);
```

### Find the Error

There are a number of syntax errors in the following program. Locate as many as you can.

```
*/  What's wrong with this program? /*
public MyProgram
{
   public static void main(String[] args);
   }
     int a, b, c  \\ Three integers
     a = 3
     b = 4
     c = a + b
     System.out.println('The value of c is' + C);
   {
```

### Algorithm Workbench

1. Show how the `double` variables `temp`, `weight`, and `age` can be declared in one statement.
2. Show how the `int` variables `months`, `days`, and `years` may be declared in one statement, with `months` initialized to 2 and `years` initialized to 3.
3. Write assignment statements that perform the following operations with the variables `a`, `b`, and `c`.
   a. Adds 2 to `a` and stores the result in `b`
   b. Multiplies `b` times 4 and stores the result in `a`
   c. Divides `a` by 3.14 and stores the result in `b`
   d. Subtracts 8 from `b` and stores the result in `a`
   e. Stores the character 'K' in `c`
   f. Stores the Unicode code for 'B' in `c`
4. Assume the variables `result`, `w`, `x`, `y`, and `z` are all integers, and that `w = 5`, `x = 4`, `y = 8`, and `z = 2`. What value will be stored in `result` in each of the following statements?
   a. `result = x + y;`
   b. `result = z * 2;`
   c. `result = y / x;`
   d. `result = y - z;`
   e. `result = w % 2;`
5. How would each of the following numbers be represented in E notation?
   a. $3.287 \times 10^6$
   b. $-9.7865 \times 10^{12}$
   c. $7.65491 \times 10^{-3}$

6. Modify the following program so it prints two blank lines between each line of text.

```
public class
{
  public static void main(String[] args)
   {
      System.out.print("Hearing in the distance");
      System.out.print("Two mandolins like creatures in the");
      System.out.print("dark");
      System.out.print("Creating the agony of ecstasy.");
      System.out.println("                     - George Barker");
   }
}
```

7. What will the following code output?

```
int apples = 0, bananas = 2, pears = 10;
apples += 10;
bananas *= 10;
pears /= 10;
System.out.println(apples + " " +
                   bananas + " " +
                   pears);
```

8. What will the following code output?

```
double d = 12.9;
int i = (int)d;
System.out.println(i);
```

9. What will the following code output?

```
String message = "Have a great day!";
System.out.println(message.charAt(5));
```

10. What will the following code output?

```
String message = "Have a great day!";
System.out.println(message.toUpperCase());
System.out.println(message);
```

11. Convert the following pseudocode to Java code. Be sure to declare the appropriate variables.

```
Store 20 in the speed variable.
Store 10 in the time variable.
Multiply speed by time and store the result in the distance variable.
Display the contents of the distance variable.
```

12. Convert the following pseudocode to Java code. Be sure to declare the appropriate variables.

```
Store 172.5 in the force variable.
Store 27.5 in the area variable.
Divide area by force and store the result in the pressure variable.
Display the contents of the pressure variable.
```

13. Write the code to set up all the necessary objects for reading keyboard input. Then write code that asks the user to enter his or her desired annual income. Store the input in a `double` variable.
14. Write the code to display a dialog box that asks the user to enter his or her desired annual income. Store the input in a `double` variable.
15. A program has a `float` variable named `total` and a `double` variable named `number`. Write a statement that assigns `number` to `total` without causing an error when compiled.

### Short Answer

1. Is the following comment a single-line style comment or a multi-line style comment?

```
/* This program was written by M. A. Codewriter */
```

2. Is the following comment a single-line style comment or a multi-line style comment?

```
// This program was written by M. A. Codewriter
```

3. Describe what the phrase “self-documenting program” means.
4. What is meant by “case-sensitive”? Why is it important for a programmer to know that Java is a case-sensitive language?
5. Briefly explain how the `print` and `println` methods are related to the `System` class and the `out` object.
6. What does a variable declaration tell the Java compiler about a variable?
7. Why are variable names like `x` not recommended?
8. What things must be considered when deciding on a data type to use for a variable?
9. Briefly describe the difference between variable assignment and variable initialization.
10. What is the difference between comments that start with the `//` characters and comments that start with the `/*` characters?
11. Briefly describe what programming style means. Why should your programming style be consistent?
12. Assume that a program uses the named constant `PI` to represent the value 3.14. The program uses the named constant in several statements. What is the advantage of using the named constant instead of the actual value 3.14 in each statement?
13. Assume the file *SalesAverage.java* is a Java source file that contains documentation comments. Assuming you are in the same folder or directory as the source code file, what command would you enter at the operating system command prompt to generate the HTML documentation files?
14. An expression adds a `byte` variable and a `short` variable. Of what data type will the result be?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Name, Age, and Annual Income

Write a program that declares the following:

- a `String` variable named `name`
- an `int` variable named `age`
- a `double` variable named `annualPay`

Store your age, name, and desired annual income as literals in these variables. The program should display these values on the screen in a manner similar to the following:

```
My name is Joe Mahoney, my age is 26 and
I hope to earn $100000.0 per year.
```

### 2. Name and Initials

Write a program that has the following `String` variables: `firstName`, `middleName`, and `lastName`. Initialize these with your first, middle, and last names. The program should also have the following `char` variables: `firstInitial`, `middleInitial`, and `lastInitial`. Store your first, middle, and last initials in these variables. The program should display the contents of these variables on the screen.

### 3. Personal Information

Write a program that displays the following information, each on a separate line:

- Your name
- Your address, with city, state, and ZIP
- Your telephone number
- Your college major

Although these items should be displayed on separate output lines, use only a single `println` statement in your program.

### 4. Star Pattern

Write a program that displays the following pattern:

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

### 5. Sales Prediction

The East Coast sales division of a company generates 62 percent of total sales. Based on that percentage, write a program that will predict how much the East Coast division will generate if the company has $4.6 million in sales this year. *Hint: Use the value 0.62 to represent 62 percent.*

### 6. Land Calculation

One acre of land is equivalent to 43,560 square feet. Write a program that calculates the number of acres in a tract of land with 389,767 square feet. *Hint: Divide the size of the tract of land by the size of an acre to get the number of acres.*

### 7. Sales Tax

Write a program that will ask the user to enter the amount of a purchase. The program should then compute the state and county sales tax. Assume the state sales tax is 4 percent and the county sales tax is 2 percent. The program should display the amount of the purchase, the state sales tax, the county sales tax, the total sales tax, and the total of the sale (which is the sum of the amount of purchase plus the total sales tax). *Hint: Use the value 0.02 to represent 2 percent, and 0.04 to represent 4 percent.*

### 8. Cookie Calories

A bag of cookies holds 40 cookies. The calorie information on the bag claims that there are 10 servings in the bag and that a serving equals 300 calories. Write a program that lets the user enter the number of cookies he or she actually ate and then reports the number of total calories consumed.

VideoNote
The Miles-per-Gallon Problem

### 9. Miles-per-Gallon

A car's miles-per-gallon (MPG) can be calculated with the following formula:

$$\mathit{MPG} = \mathit{Miles\ driven}/\mathit{Gallons\ of\ gas\ used}$$

Write a program that asks the user for the number of miles driven and the gallons of gas used. It should calculate the car's miles-per-gallon and display the result on the screen.

### 10. Test Average

Write a program that asks the user to enter three test scores. The program should display each test score, as well as the average of the scores.

### 11. Circuit Board Profit

An electronics company sells circuit boards at a 40 percent profit. If you know the retail price of a circuit board, you can calculate its profit with the following formula:

$$\text{Profit} = \text{Retail price} \times 0.4$$

Write a program that asks the user for the retail price of a circuit board, calculates the amount of profit earned for that product, and displays the results on the screen.

### 12. String Manipulator

Write a program that asks the user to enter the name of his or her favorite city. Use a `String` variable to store the input. The program should display the following:

- The number of characters in the city name
- The name of the city in all uppercase letters
- The name of the city in all lowercase letters
- The first character in the name of the city

### 13. Restaurant Bill

Write a program that computes the tax and tip on a restaurant bill. The program should ask the user to enter the charge for the meal. The tax should be 6.75 percent of the meal charge. The tip should be 20 percent of the total after adding the tax. Display the meal charge, tax amount, tip amount, and total bill on the screen.

### 14. Male and Female Percentages

Write a program that asks the user for the number of males and the number of females registered in a class. The program should display the percentage of males and females in the class.

*Hint: Suppose there are 8 males and 12 females in a class. There are 20 students in the class. The percentage of males can be calculated as $8 \div 20 = 0.4$, or 40%. The percentage of females can be calculated as $12 \div 20 = 0.6$, or 60%.*

### 15. Stock Commission

Kathryn bought 600 shares of stock at a price of $21.77 per share. She must pay her stockbroker a 2 percent commission for the transaction. Write a program that calculates and displays the following:

- The amount paid for the stock alone (without the commission)
- The amount of the commission
- The total amount paid (for the stock plus the commission)

### 16. Energy Drink Consumption

A soft drink company recently surveyed 12,467 of its customers and found that approximately 14 percent of those surveyed purchase one or more energy drinks per week. Of those customers who purchase energy drinks, approximately 64 percent of them prefer citrus-flavored energy drinks. Write a program that displays the following:

- The approximate number of customers in the survey who purchase one or more energy drinks per week
- The approximate number of customers in the survey who prefer citrus-flavored energy drinks

### 17. Ingredient Adjuster

A cookie recipe calls for the following ingredients:

- 1.5 cups of sugar
- 1 cup of butter
- 2.75 cups of flour

The recipe produces 48 cookies with these amounts of the ingredients. Write a program that asks the user how many cookies he or she wants to make, and then displays the number of cups of each ingredient needed for the specified number of cookies.

### 18. Word Game

Write a program that plays a word game with the user. The program should ask the user to enter the following:

- His or her name
- His or her age

- The name of a city
- The name of a college
- A profession
- A type of animal
- A pet's name

After the user has entered these items, the program should display the following story, inserting the user's input into the appropriate locations:

> There once was a person named ***NAME*** who lived in ***CITY***. At the age of ***AGE***, ***NAME*** went to college at ***COLLEGE***. ***NAME*** graduated and went to work as a ***PROFESSION***. Then, ***NAME*** adopted a(n) ***ANIMAL*** named ***PETNAME***. They both lived happily ever after!

## 19. Stock Transaction Program

*Last month Joe purchased some stock in Acme Software, Inc. Here are the details of the purchase:*

- The number of shares that Joe purchased was 1,000.
- When Joe purchased the stock, he paid $32.87 per share.
- Joe paid his stockbroker a commission that amounted to 2% of the amount he paid for the stock.

Two weeks later Joe sold the stock. Here are the details of the sale:

- The number of shares that Joe sold was 1,000.
- He sold the stock for $33.92 per share.
- He paid his stockbroker another commission that amounted to 2% of the amount he received for the stock.

Write a program that displays the following information:

- The amount of money Joe paid for the stock.
- The amount of commission Joe paid his broker when he bought the stock.
- The amount that Joe sold the stock for.
- The amount of commission Joe paid his broker when he sold the stock.
- Display the amount of profit that Joe made after selling his stock and paying the two commissions to his broker. (If the amount of profit that your program displays is a negative number, then Joe lost money on the transaction.)

## 20. Planting Grapevines

A vineyard owner is planting several new rows of grapevines and needs to know how many grapevines to plant in each row. She has determined that after measuring the length of a future row, she can use the following formula to calculate the number of vines that will fit in the row, along with the trellis end-post assemblies that will need to be constructed at each end of the row:

$$V = \frac{R - 2E}{S}$$

The terms in the formula are:

$V$ is the number of grapevines that will fit in the row.
$R$ is the length of the row, in feet.
$E$ is the amount of space used by an end-post assembly.
$S$ is the space between vines, in feet.

Write a program that makes the calculation for the vineyard owner. The program should ask the user to input the following:

- The length of the row, in feet
- The amount of space used by an end-post assembly, in feet
- The amount of space between the vines, in feet

Once the input data has been entered, the program should calculate and display the number of grapevines that will fit in the row.

### 21. Compound Interest

When a bank account pays compound interest, it pays interest not only on the principal amount that was deposited into the account, but also on the interest that has accumulated over time. Suppose you want to deposit some money into a savings account, and let the account earn compound interest for a certain number of years. The formula for calculating the balance of the account after a specified number of years is:

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

The terms in the formula are:

> $A$ is the amount of money in the account after the specified number of years.
> $P$ is the principal amount that was originally deposited into the account.
> $r$ is the annual interest rate.
> $n$ is the number of times per year that the interest is compounded.
> $t$ is the specified number of years.

Write a program that makes the calculation for you. The program should ask the user to input the following:

- The amount of principal originally deposited into the account
- The annual interest rate paid by the account
- The number of times per year that the interest is compounded (For example, if interest is compounded monthly, enter 12. If interest is compounded quarterly, enter 4.)
- The number of years the account will be left to earn interest

Once the input data has been entered, the program should calculate and display the amount of money that will be in the account after the specified number of years.

**NOTE:** *The user should enter the interest rate as a percentage. For example, 2 percent would be entered as 2, not as .02. The program will then have to divide the input by 100 to move the decimal point to the correct position.*

# CHAPTER 3

# Decision Structures

## TOPICS

3.1 The `if` Statement
3.2 The `if-else` Statement
3.3 Nested `if` Statements
3.4 The `if-else-if` Statement
3.5 Logical Operators
3.6 Comparing `String` Objects
3.7 More about Variable Declaration and Scope
3.8 The Conditional Operator (Optional)
3.9 The `switch` Statement
3.10 Displaying Formatted Output with `System.out.printf` and `String.format`
3.11 Common Errors to Avoid

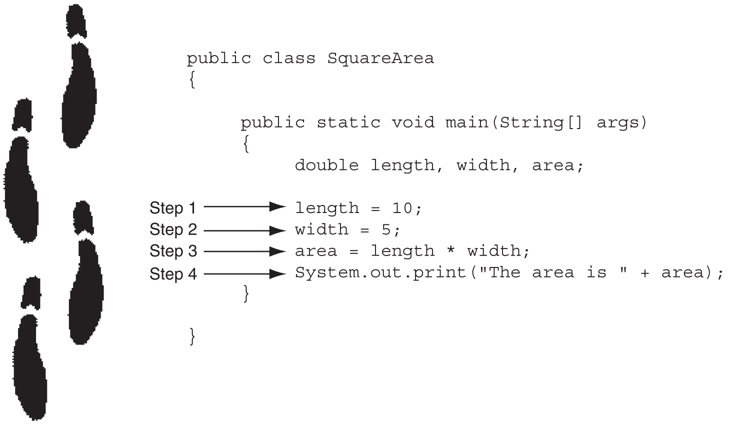
## 3.1 The `if` Statement

**CONCEPT:** **The `if` statement is used to create a decision structure, which allows a program to have more than one path of execution. The `if` statement causes one or more statements to execute only when a `boolean` expression is `true`.**

VideoNote
The `if` Statement

In all the programs you have written so far, the statements are executed one after the other, in the order they appear. You might think of sequentially executed statements as the steps you take as you walk down a road. To complete the journey, you must start at the beginning and take each step, one after the other, until you reach your destination. This is illustrated in Figure 3-1.

**Figure 3-1** Sequence structure

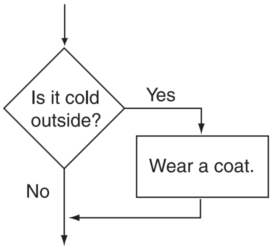


```
public class SquareArea
{

     public static void main(String[] args)
     {
          double length, width, area;

          length = 10;
          width = 5;
          area = length * width;
          System.out.print("The area is " + area);
     }

}
```

The type of code shown in Figure 3-1 is called a *sequence structure*, because the statements are executed in sequence, without branching off in another direction. Programs often need more than one path of execution, however. Many algorithms require a program to execute some statements only under certain circumstances. This can be accomplished with a *decision structure*.

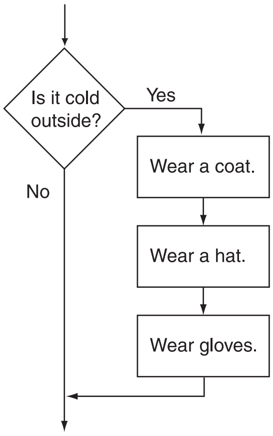
In a decision structure's simplest form, a specific action is taken only when a condition exists. If the condition does not exist, the action is not performed. The flowchart in Figure 3-2 shows the logic of a decision structure. The diamond symbol represents a yes/no question or a true/false condition. If the answer to the question is yes (or if the condition is true), the program flow follows one path, which leads to an action being performed. If the answer to the question is no (or the condition is false), the program flow follows another path, which skips the action.

**Figure 3-2** Simple decision structure logic



In the flowchart, the action "Wear a coat" is performed only when it is cold outside. If it is not cold outside, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition (cold outside) exists. Figure 3-3 shows a more elaborate flowchart, where three actions are taken only when it is cold outside.

**Figure 3-3** Three-action decision structure logic



One way to code a decision structure in Java is with the `if` statement. Here is the general format of the `if` statement:

```
if (BooleanExpression)
    statement;
```

The `if` statement is simple in the way it works: The *`BooleanExpression`* that appears inside the parentheses must be a `boolean` expression. A *`boolean` expression* is one that is either `true` or `false`. If the `boolean` expression is `true`, the very next `statement` is executed. Otherwise, it is skipped. The `statement` is conditionally executed because it executes only under the condition that the expression in the parentheses is `true`.

## Using Relational Operators to Form Conditions

Typically, the `boolean` expression that is tested by an `if` statement is formed with a relational operator. A *relational operator* determines whether a specific relationship exists between two values. For example, the greater than operator (`>`) determines whether one value is greater than another. The equal to operator (`==`) determines whether two values are equal. Table 3-1 lists all of the Java relational operators.

**Table 3-1** Relational operators

| Relational Operators (in Order of Precedence) | Meaning |
|---|---|
| `>` | Greater than |
| `<` | Less than |
| `>=` | Greater than or equal to |
| `<=` | Less than or equal to |
| `==` | Equal to |
| `!=` | Not equal to |

All of the relational operators are binary, which means they use two operands. Here is an example of an expression using the greater than operator:

```
length > width
```

This expression determines whether `length` is greater than `width`. If `length` is greater than `width`, the value of the expression is `true`. Otherwise, the value of the expression is `false`. Because the expression can be only `true` or `false`, it is a `boolean` expression. The following expression uses the less than operator to determine whether `length` is less than `width`:

```
length < width
```

Table 3-2 shows examples of several `boolean` expressions that compare the variables `x` and `y`.

**Table 3-2** `boolean` expressions using relational operators

| Expression | Meaning |
|---|---|
| `x > y` | Is `x` greater than `y`? |
| `x < y` | Is `x` less than `y`? |
| `x >= y` | Is `x` greater than or equal to `y`? |
| `x <= y` | Is `x` less than or equal to `y`? |
| `x == y` | Is `x` equal to `y`? |
| `x != y` | Is `x` not equal to `y`? |

Two of the operators, `>=` and `<=`, test for more than one relationship. The `>=` operator determines whether the operand on its left is greater than or equal to the operand on its right. Assuming that `a` is 4, `b` is 6, and `c` is 4, both of the expressions `b >= a` and `a >= c` are `true`, but `a >= 5` is `false`. When using this operator, the `>` symbol must precede the `=` symbol, and there is no space between them. The `<=` operator determines whether the operand on its left is less than or equal to the operand on its right. Once again, assuming that `a` is 4, `b` is 6, and `c` is 4, both `a <= c` and `b <= 10` are `true`, but `b <= a` is `false`. When using this operator, the `<` symbol must precede the `=` symbol, and there is no space between them.

The `==` operator determines whether the operand on its left is equal to the operand on its right. If both operands have the same value, the expression is `true`. Assuming that `a` is 4, the expression `a == 4` is `true` and the expression `a == 2` is `false`. Notice the equality operator is two `=` symbols together. Don't confuse this operator with the assignment operator, which is one `=` symbol.

The `!=` operator is the not equal operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the `==` operator. As before, assuming `a` is 4, `b` is 6, and `c` is 4, both `a != b` and `b != c` are `true` because `a` is not equal to `b` and `b` is not equal to `c`. However, `a != c` is `false` because `a` is equal to `c`.

## Putting It All Together

Let's look at an example of the `if` statement:

```
if (sales > 50000)
    bonus = 500.0;
```

This statement uses the > operator to determine whether `sales` is greater than 50,000. If the expression `sales > 50000` is `true`, the variable `bonus` is assigned 500.0. If the expression is `false`, however, the assignment statement is skipped. The program in Code Listing 3-1 shows another example. The user enters three test scores and the program calculates their average. If the average is greater than 95, the program congratulates the user on obtaining a high score.

**Code Listing 3-1** `(AverageScore.java)`

```
import java.util.Scanner;

/**
  This program demonstrates the if statement.
*/

public class AverageScore
{
   public static void main(String[] args)
   {
      double score1,   // Score #1
             score2,   // Score #2
             score3,   // Score #3
             average;  // Average score

      // Create a Scanner object to read input.
      Scanner keyboard = new Scanner(System.in);

      System.out.println("This program averages 3 test scores.");

      // Get the first score.
      System.out.print("Enter score #1: ");
      score1 = keyboard.nextDouble();

      // Get the second score.
      System.out.print("Enter score #2: ");
      score2 = keyboard.nextDouble ();

      // Get the third score.
      System.out.print("Enter score #3: ");
      score3 = keyboard.nextDouble ();

      // Calculate and display the average score.
      average = (score1 + score2 + score3) / 3.0;
      System.out.println("The average is " + average);

      // If the average is higher than 95, congratulate the user.
      if (average > 95)
         System.out.println("That's a great score!");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
This program averages 3 test scores.
Enter score #1: 82 [Enter]
Enter score #2: 76 [Enter]
Enter score #3: 91 [Enter]
The average is 83.0
```

**Program Output with Example Input Shown in Bold**

```
This program averages 3 test scores.
Enter score #1: 97 [Enter]
Enter score #2: 94 [Enter]
Enter score #3: 100 [Enter]
The average is 97.0
That's a great score!
```

The code in lines 38 and 39 causes the congratulatory message to be printed:

```
if (average > 95)
    System.out.println("That's a great score!");
```

Figure 3-4 shows the logic of this `if` statement.

Table 3-3 shows other examples of `if` statements and their outcomes.

**Figure 3-4** Logic of the `if` statement

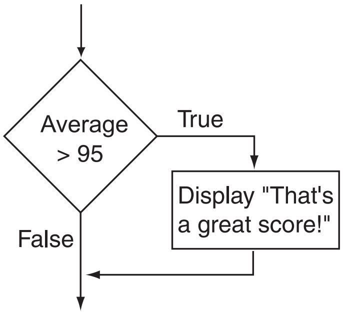


**Table 3-3** Other examples of `if` statements

| Statement | Outcome |
|---|---|
| `if (hours > 40)`<br>`   overTime = true;` | If `hours` is greater than 40, assigns `true` to the `boolean` variable `overTime`. |
| `if (value < 32)`<br>`   System.out.println("Invalid number");` | If `value` is less than 32, displays the message `"Invalid number"`. |

## Programming Style and the `if` Statement

Even though an `if` statement usually spans more than one line, it is really one long statement. For instance, the following `if` statements are identical except for the style in which they are written:

```
if (value > 32)
   System.out.println("Invalid number.");
if (value > 32) System.out.println("Invalid number.");
```

In both of these examples, the compiler considers the `if` statement and the conditionally executed statement as one unit, with a semicolon properly placed at the end. Indentions and spacing are for the human readers of a program, not the compiler. Here are two important style rules you should adopt for writing `if` statements:

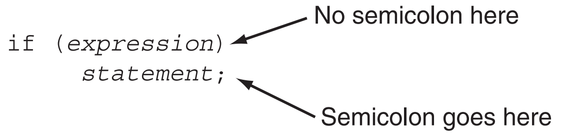
- The conditionally executed statement should appear on the line after the `if` statement.
- The conditionally executed statement should be indented one level from the `if` statement.

In most editors, each time you press the tab key, you are indenting one level. By indenting the conditionally executed statement, you are causing it to stand out visually. This is so you can tell at a glance what part of the program the `if` statement executes. This is a standard way of writing `if` statements and is the method you should use.

## Be Careful with Semicolons

You do not put a semicolon after the `if (`*`expression`*`)` portion of an `if` statement, as illustrated in Figure 3-5. This is because the `if` statement isn't complete without its conditionally executed statement.

**Figure 3-5** Do not prematurely terminate an `if` statement with a semicolon



```
if (expression)
      statement;
```

If you prematurely terminate an `if` statement with a semicolon, the compiler will not display an error message, but will assume that you are placing a *null statement* there. The null statement, which is an empty statement that does nothing, will become the conditionally executed statement. The statement that you intended to be conditionally executed will be disconnected from the `if` statement and will always execute.

For example, look at the following code:

```
int x = 0, y = 10;


// The following if statement is prematurely
// terminated with a semicolon.
if (x > y);
   System.out.println(x + " is greater than " + y);
```

The `if` statement in this code is prematurely terminated with a semicolon. Because the `println` statement is not connected to the `if` statement, it will always execute.

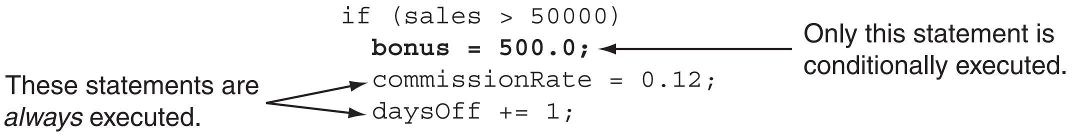
## Having Multiple Conditionally Executed Statements

The previous examples of the `if` statement conditionally execute a single statement. The `if` statement can also conditionally execute a group of statements, as long as they are enclosed in a set of braces. Enclosing a group of statements inside braces creates a *block* of statements. Here is an example:

```
if (sales > 50000)
{
     bonus = 500.0;
     commissionRate = 0.12;
     daysOff += 1;
}
```

If `sales` is greater than 50,000, this code will execute all three of the statements inside the braces, in the order they appear. If the braces are accidentally left out, however, the `if` statement conditionally executes only the very next statement. Figure 3-6 illustrates this.

**Figure 3-6** An `if` statement missing its braces



```
if (sales > 50000)
   bonus = 500.0;
   commissionRate = 0.12;
   daysOff += 1;
```

Only this statement is conditionally executed.

These statements are *always* executed.

## Flags

A flag is a `boolean` variable that signals when some condition exists in the program. When the flag variable is set to `false`, it indicates the condition does not yet exist. When the flag variable is set to `true`, it means the condition does exist.

For example, suppose a program similar to the previous test averaging program has a `boolean` variable named `highScore`. The variable might be used to signal that a high score has been achieved by the following code:

```
if (average > 95)
   highScore = true;
```

Later, the same program might use code similar to the following to test the `highScore` variable, in order to determine whether a high score has been achieved:

```
if (highScore)
   System.out.println("That's a high score!");
```

You will find flag variables useful in many circumstances, and we will come back to them in future chapters.

## Comparing Characters

You can use the relational operators to test character data as well as numbers. For example, assuming that `ch` is a `char` variable, the following code segment uses the == operator to compare it to the character 'A':

```
if (ch == 'A')
   System.out.println("The letter is A.");
```

The != operator can also be used with characters to test for inequality. For example, the following statement determines whether the `char` variable `ch` is not equal to the letter 'A':

```
if (ch != 'A')
   System.out.println("Not the letter A.");
```

You can also use the >, <, >=, and <= operators to compare characters. Computers do not actually store characters, such as A, B, C, and so forth, in memory. Instead, they store numeric codes that represent the characters. Recall from Chapter 2 that Java uses Unicode, which is a set of numbers that represents all the letters of the alphabet (both lowercase and uppercase), the printable digits 0 through 9, punctuation symbols, and special characters. When a character is stored in memory, it is actually the Unicode number that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.

**NOTE:** Unicode is an international encoding system that is extensive enough to represent all the characters of all the world's alphabets.

In Unicode, letters are arranged in alphabetic order. Because 'A' comes before 'B', the numeric code for the character 'A' is less than the code for the character 'B'. (The code for 'A' is 65 and the code for 'B' is 66. Appendix A, available for download from this book's companion Web site, lists the codes for all of the printable English characters.) In the following `if` statement, the `boolean` expression `'A' < 'B'` is `true`:

```
if ('A' < 'B')
   System.out.println("A is less than B.");
```

In Unicode, the uppercase letters come before the lowercase letters, so the numeric code for 'A' (65) is less than the numeric code for 'a' (97). In addition, the space character (code 32) comes before all the alphabetic characters.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.1 Write an `if` statement that assigns 0 to `x` when `y` is equal to 20.

3.2 Write an `if` statement that multiplies `payRate` by 1.5 if `hours` is greater than 40.

3.3 Write an `if` statement that assigns 0.2 to `commission` if `sales` is greater than or equal to 10000.

3.4 Write an `if` statement that sets the variable `fees` to 50 if the `boolean` variable `max` is `true`.

3.5 Write an `if` statement that assigns 20 to the variable `y` and assigns 40 to the variable `z` if the variable `x` is greater than 100.

3.6 Write an `if` statement that assigns 0 to the variable `b` and assigns 1 to the variable `c` if the variable `a` is less than 10.

3.7 Write an `if` statement that displays "Goodbye" if the variable `myCharacter` contains the character `'D'`.

## 3.2 The `if-else` Statement

**CONCEPT: The `if-else` statement will execute one group of statements if its `boolean` expression is `true`, or another group if its `boolean` expression is `false`.**

VideoNote
The `if-else` Statement

The `if-else` statement is an expansion of the `if` statement. Here is its format:

```
if (BooleanExpression)
    statement or block
else
    statement or block
```

Like the `if` statement, a `boolean` expression is evaluated. If the expression is `true`, a statement or block of statements is executed. If the expression is `false`, however, a separate group of statements is executed. The program in Code Listing 3-2 uses the `if-else` statement to handle a classic programming problem: division by zero. In Java, a program crashes when it divides an integer by 0. When a floating-point value is divided by 0, the program doesn't crash. Instead, the special value `Infinity` is produced as the result of the division.

**Code Listing 3-2** (`Division.java`)

```
import java.util.Scanner; // Needed for the Scanner class

/**
   This program demonstrates the if-else statement.
*/

public class Division
{
   public static void main(String[] args)
   {
      double number1, number2; // Division operands
      double quotient;         // Result of division

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the first number.
      System.out.print("Enter a number: ");
      number1 = keyboard.nextDouble();

      // Get the second number.
```

```
    System.out.print("Enter another number: ");
    number2 = keyboard.nextDouble();

    if (number2 == 0)
    {
        System.out.println("Division by zero is not possible.");
        System.out.println("Please run the program again and ");
        System.out.println("enter a number other than zero.");
    }
    else
    {
        quotient = number1 / number2;
        System.out.print("The quotient of " + number1);
        System.out.print(" divided by " + number2);
        System.out.println(" is " + quotient);
    }
  }
}
```

**Program Output with Example Input Shown in Bold**

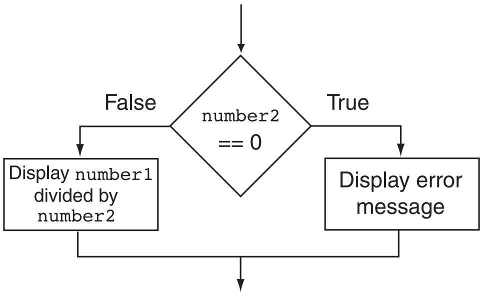
```
Enter a number: 10 [Enter]
Enter another number: 0 [Enter]
Division by zero is not possible.
Please run the program again and
enter a number other than zero.
```

**Program Output with Example Input Shown in Bold**

```
Enter a number: 10 [Enter]
Enter another number: 5 [Enter]
The quotient of 10 divided by 5 is 2.0
```

The value of `number2` is tested before the division is performed. If the user enters 0, the block of statements controlled by the `if` clause executes, displaying a message that indicates the program cannot perform division by zero. Otherwise, the `else` clause takes control, which divides `number1` by `number2` and displays the result. Figure 3-7 shows the logic of the `if-else` statement.

**Figure 3-7** Logic of the `if-else` statement



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

3.8 Write an `if-else` statement that assigns 20 to the variable `y` if the variable `x` is greater than 100. Otherwise, it should assign 0 to the variable `y`.

3.9 Write an `if-else` statement that assigns 1 to `x` when `y` is equal to 100. Otherwise, it should assign 0 to `x`.

3.10 Write an `if-else` statement that assigns 0.10 to `commission` unless `sales` is greater than or equal to 50000.0, in which case it assigns 0.2 to `commission`.

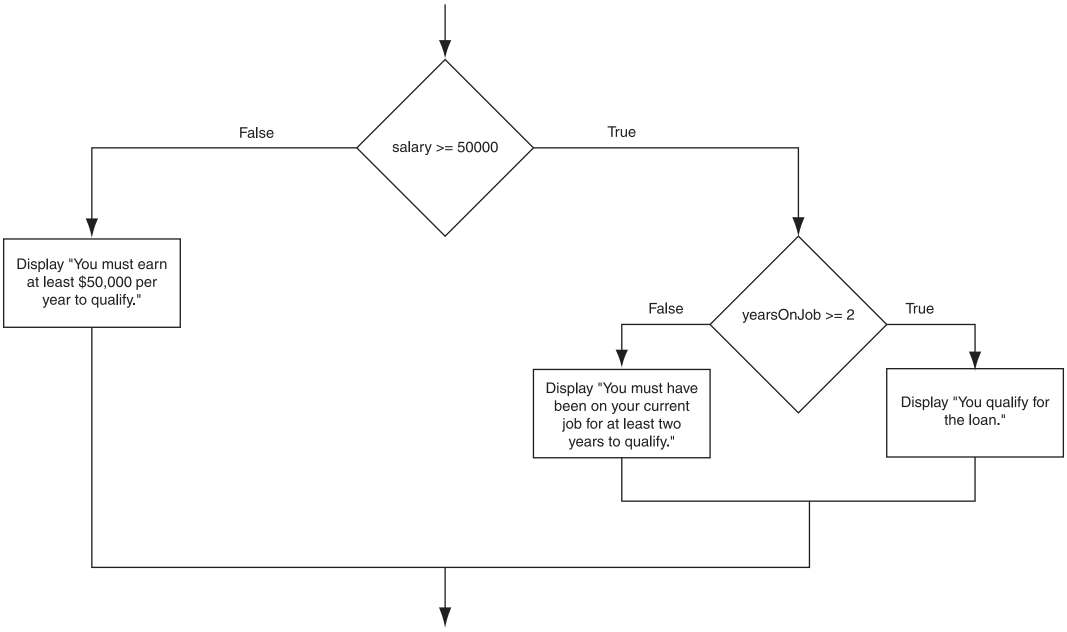
3.11 Write an `if-else` statement that assigns 0 to the variable `b` and assigns 1 to the variable `c` if the variable `a` is less than 10. Otherwise, it should assign −99 to the variable `b` and assign 0 to the variable `c`.

## 3.3 Nested `if` Statements

**CONCEPT: To test more than one condition, an `if` statement can be nested inside another `if` statement.**

Sometimes an `if` statement must be nested inside another `if` statement. For example, consider a banking program that determines whether a bank customer qualifies for a special, low interest rate on a loan. To qualify, two conditions must exist: (1) the customer's salary must be at least $50,000, and (2) the customer must have held his or her current job for at least two years. Figure 3-8 shows a flowchart for an algorithm that could be used in such a program.

**Figure 3-8** Logic of nested `if` statements



salary >= 50000
False
True
Display "You must earn at least $50,000 per year to qualify."
yearsOnJob >= 2
False
True
Display "You must have been on your current job for at least two years to qualify."
Display "You qualify for the loan."

If we follow the flow of execution in the flowchart, we see that the expression `salary >= 30000` is tested. If this expression is false, there is no need to perform further tests; we know that the customer does not qualify for the special interest rate. If the expression is true, however, we need to test the second condition. This is done with a nested decision structure that tests the expression `yearsOnJob >= 2`. If this expression is true, then the customer qualifies for the special interest rate. If this expression is false, then the customer does not qualify. Code Listing 3-3 shows the complete program.

**Code Listing 3-3** `(LoanQualifier.java)`

```java
import java.util.Scanner;

/**
   This program demonstrates a nested if statement.
*/

public class LoanQualifier
{
   public static void main(String[] args)
   {
      double salary;      // Annual salary
      double yearsOnJob;  // Years at current job

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the user's annual salary.
      System.out.print("Enter your annual salary: ");
      salary = keyboard.nextDouble();

      // Get the number of years at the current job.
      System.out.print("Enter the number of years " +
                       "at your current job: ");
      yearsOnJob = keyboard.nextDouble();

      // Determine whether the user qualifies for the loan.
      if (salary >= 50000)
      {
         if (yearsOnJob >= 2)
         {
            System.out.println("You qualify for the loan.");
         }
         else
         {
            System.out.println("You must have been on your " +
                               "current job for at least " +
                               "two years to qualify.");
         }
      }
40          else
41          {
42                System.out.println("You must earn at least " +
43                                   "$50,000 per year to qualify.");
44          }
45       }
46    }
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 55000.00 [Enter]
Enter the number of years at your current job: 1 [Enter]
You must have been on your current job for at least two years to qualify.
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 25000.00 [Enter]
Enter the number of years at your current job: 5 [Enter]
You must earn at least $50,000 per year to qualify.
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 55000.00 [Enter]
Enter the number of years at your current job: 5 [Enter]
You qualify for the loan.
```

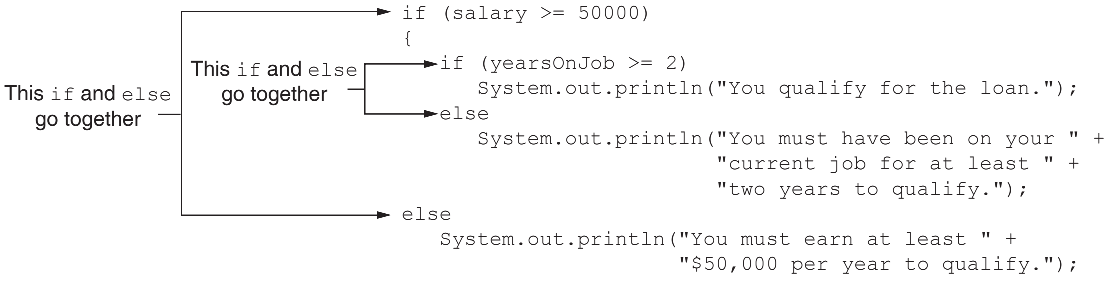
The first `if` statement (which begins in line 27) conditionally executes the second one (which begins in line 29). The only way the program will execute the second `if` statement is for the `salary` variable to contain a value that is greater than or equal to 50,000. When this is the case, the second `if` statement will test the `yearsOnJob` variable. If it contains a value that is greater than or equal to 2, a message will be displayed informing the user that he or she qualifies for the loan.

It should be noted that the braces used in the `if` statements in this program are not required. They could have been written as follows:

```
if (salary >= 50000)
{
   if (yearsOnJob >= 2)
      System.out.println("You qualify for the loan.");
   else
      System.out.println("You must have been on your " +
                         "current job for at least " +
                         "two years to qualify.");
else
      System.out.println("You must earn at least " +
                         "$50,000 per year to qualify.");
```

Not only do the braces make the statements easier to read, but they also help in debugging code. When debugging a program with nested `if-else` statements, it's important to know which `if` clause each `else` clause belongs to. The rule for matching `else` clauses with `if` clauses is this: An `else` clause goes with the closest previous `if` clause that doesn't already have its own `else` clause. This is easy to see when the conditionally executed statements are enclosed in braces and are properly indented, as shown in Figure 3-9. Each `else` clause lines up with the `if` clause it belongs to. These visual cues are important because nested `if` statements can be very long and complex.

**Figure 3-9** Alignment of `if` and `else` clauses



```
if (salary >= 50000)
{
   if (yearsOnJob >= 2)
      System.out.println("You qualify for the loan.");
   else
      System.out.println("You must have been on your " +
                         "current job for at least " +
                         "two years to qualify.");
else
   System.out.println("You must earn at least " +
                      "$50,000 per year to qualify.");
```

### Testing a Series of Conditions

In the previous example, you saw how a program can use nested decision structures to test more than one condition. It is not uncommon for a program to have a series of conditions to test, and then perform an action depending on which condition is true. One way to accomplish this it to have a decision structure with numerous other decision structures nested inside it. For example, consider the program presented in the following *In the Spotlight* section.

## In the Spotlight:

## Multiple Nested Decision Structures

Suppose one of your professors uses the following 10-point grading scale for exams:

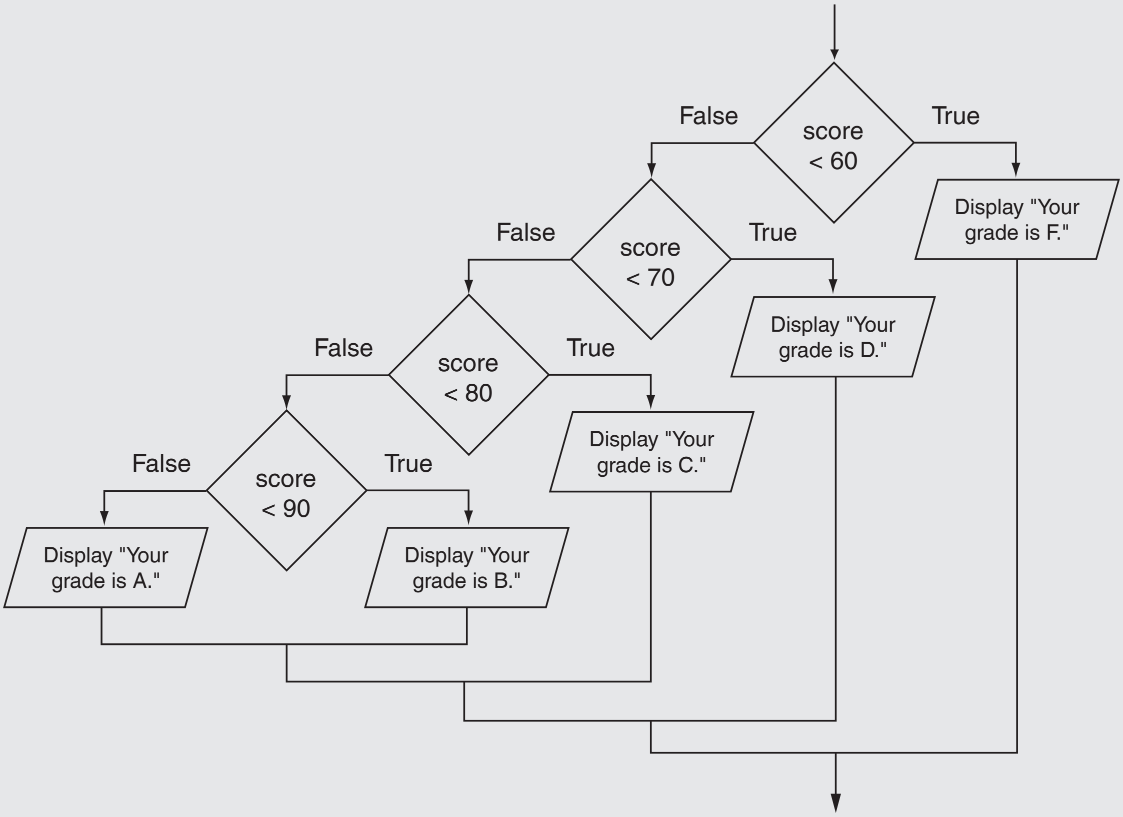
| Test Score | Grade |
|---|---|
| 90 and above | A |
| 80–89 | B |
| 70–79 | C |
| 60–69 | D |
| Below 60 | F |

Your professor has asked you to write a program that will allow a student to enter a test score and then display the grade for that score. Here is the algorithm that you will use:

```
Ask the user to enter a test score.
Determine the grade in the following manner:
If the score is less than 60, then the grade is F.
  Otherwise, if the score is less than 70, then the grade is D.
    Otherwise, if the score is less than 80, then the grade is C.
      Otherwise, if the score is less than 90, then the grade is B.
        Otherwise, the grade is A.
```

You decide that the process of determining the grade will require several nested decision structures, as shown in Figure 3-10. Code Listing 3-4 shows the complete program. The code for the nested decision structures is in lines 23 through 51. Figures 3-11 and 3-12 show what happens in two different sessions with the program.

**Figure 3-10** Nested decision structure to determine a grade


score < 60
False
True
Display "Your grade is F."
score < 70
False
True
Display "Your grade is D."
score < 80
False
True
Display "Your grade is C."
score < 90
False
True
Display "Your grade is A."
Display "Your grade is B."

**Code Listing 3-4** (NestedDecision.java)

```
1 import javax.swing.JOptionPane; // Needed for JOptionPane
2
3  /**
4     This program asks the user to enter a numeric test
5     score and displays a letter grade for the score. The
6     program uses nested decision structures
7     to determine the grade.
8  */
9
```

```
public class NestedDecision
{
    public static void main(String[] args)
    {
       int testScore;  // Numeric test score
       String input;   // To hold the user's input

       // Get the numeric test score.
       input = JOptionPane.showInputDialog("Enter your numeric " +
                      "test score and I will tell you the grade: ");
       testScore = Integer.parseInt(input);

       // Display the grade.
       if (testScore < 60)
       {
          JOptionPane.showMessageDialog(null, "Your grade is F.");
       }
       else
       {
          if (testScore < 70)
          {
            JOptionPane.showMessageDialog(null, "Your grade is D.");
          }
          else
          {
            if (testScore < 80)
            {
              JOptionPane.showMessageDialog(null, "Your grade is C.");
            }
            else
            {
              if (testScore < 90)
              {
                JOptionPane.showMessageDialog(null, "Your grade is B.");
              }
              else
              {
                JOptionPane.showMessageDialog(null, "Your grade is A.");
              }
            }
          }
       }
       System.exit(0);
    }
}
```

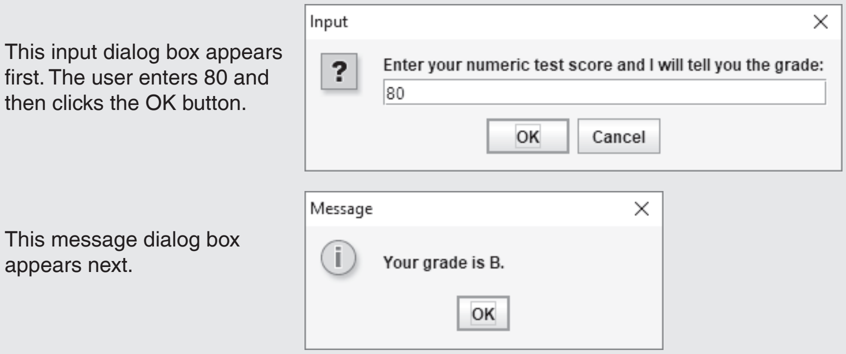


**Figure 3-11** Interaction with the `NestedDecision` program (Oracle Corporate Counsel)

This input dialog box appears first. The user enters 80 and then clicks the OK button.

Input

Enter your numeric test score and I will tell you the grade:

80

This message dialog box appears next.

Message

Your grade is B.

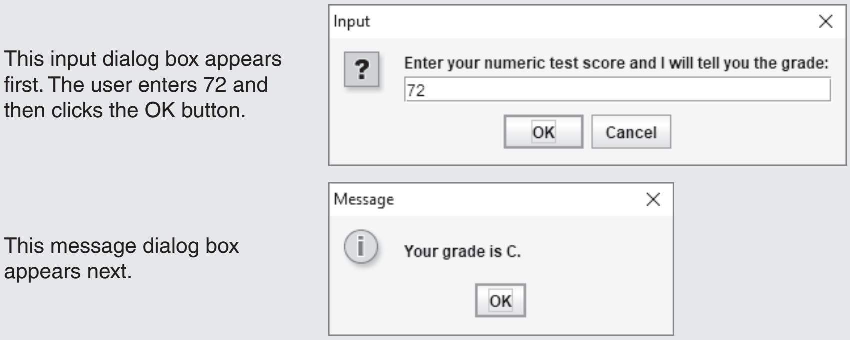


**Figure 3-12** Interaction with the `NestedDecision` program (Oracle Corporate Counsel)

This input dialog box appears first. The user enters 72 and then clicks the OK button.

Input

Enter your numeric test score and I will tell you the grade:

72

This message dialog box appears next.

Message

Your grade is C.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.12 Write nested `if` statements that perform the following test: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of the two.

3.13 Write code that tests the variable `x` to determine whether it is greater than 0. If `x` is greater than 0, the code should test the variable `y` to determine whether it is less than 20. If `y` is less than 20, the code should assign 1 to the variable `z`. If `y` is not less than 20, the code should assign 0 to the variable `z`.

## 3.4 The `if-else-if` Statement

**CONCEPT:** **The `if-else-if` statement tests a series of conditions. It is often simpler to test a series of conditions with the `if-else-if` statement than with a set of nested `if-else` statements.**

VideoNote
The `if-else-if` Statement

Even though Code Listing 3-4 is a simple example, the logic of the nested decision structure is fairly complex. You can alternatively test a series of conditions using the `if-else-if` statement. The `if-else-if` statement makes certain types of nested decision logic simpler to write. Here is the general format of the `if-else-if` statement:

```
if (expression_1)
{
  statement                      } If expression_1 is true these
  statement                      } statements are executed, and the rest
  etc.                           } of the structure is ignored.
}
else if (expression_2)
{
  statement                      } Otherwise, if expression_2 is true
  statement                      } these statements are executed, and
  etc.                           } the rest of the structure is ignored.
}
Insert as many else if clauses as necessary
else
{
  statement                      } These statements are executed if none
  statement                      } of the expressions above are true.
  etc.
}
```

When the statement executes, *`expression_1`* is tested. If *`expression_1`* is true, the block of statements that immediately follows is executed, and the rest of the structure is ignored. If *`expression_1`* is false, however, the program jumps to the next `else if` clause and tests *`expression_2`*. If it is true, the block of statements that immediately follows is executed, and then the rest of the structure is ignored. This process continues, from the top of the structure to the bottom, until one of the expressions is found to be true. If none of the expressions are true, the last `else` clause takes over and the block of statements immediately following it is executed.

The last `else` clause, which does not have an `if` statement following it, is referred to as the *trailing* `else`. The trailing `else` is optional, but in most cases you will use it.

**NOTE:** The general format shows braces surrounding each block of conditionally executed statements. As with other forms of the `if` statement, the braces are required only when more than one statement is conditionally executed.

Code Listing 3-5 shows an example of the `if-else-if` statement. This program is a modification of Code Listing 3-4, which appears in the previous *In the Spotlight* section. The output of this program is the same as Code Listing 3-4.

**Code Listing 3-5** (TestResults.Java)

```
 1 import javax.swing.JOptionPane;    // Needed for JOptionPane
 2
 3 /**
 4    This program asks the user to enter a numeric test
 5    score and displays a letter grade for the score. The
 6    program uses an if-else-if statement to determine
 7    the letter grade.
 8 */
 9
10 public class TestResults
11 {
12   public static void main(String[] args)
13   {
14     int testScore;    // Numeric test score
15     String input;     // To hold the user's input
16
17     // Get the numeric test score.
18     input = JOptionPane.showInputDialog("Enter your numeric " +
19                    "test score and I will tell you the grade: ");
20     testScore = Integer.parseInt(input);
21
22     // Display the grade.
23     if (testScore < 60)
24        JOptionPane.showMessageDialog(null, "Your grade is F.");
25     else if (testScore < 70)
26        JOptionPane.showMessageDialog(null, "Your grade is D.");
27     else if (testScore < 80)
28        JOptionPane.showMessageDialog(null, "Your grade is C.");
29     else if (testScore < 90)
30        JOptionPane.showMessageDialog(null, "Your grade is B.");
31     else
32        JOptionPane.showMessageDialog(null, "Your grade is A.");
33
34     System.exit(0);
35   }
36 }
```

Let's analyze how the `if-else-if` statement in lines 23 through 32 works. First, the expression `testScore < 60` is tested in line 23:

```
→ if (testScore < 60)
    JOptionPane.showMessageDialog(null, "Your grade is F.");
  else if (testScore < 70)
    JOptionPane.showMessageDialog(null, "Your grade is D.");
  else if (testScore < 80)
  JOptionPane.showMessageDialog(null, "Your grade is C.");
else if (testScore < 90)
  JOptionPane.showMessageDialog(null, "Your grade is B.");
else
  JOptionPane.showMessageDialog(null, "Your grade is A.");
```

If `testScore` is less than 60, the message `"Your grade is F."` is displayed and the rest of the `if-else-if` statement is skipped. If `testScore` is not less than 60, the `else` clause in line 25 takes over and causes the next `if` statement to be executed:

```
    if (testScore < 60)
      JOptionPane.showMessageDialog(null, "Your grade is F.");
 →  else if (testScore < 70)
      JOptionPane.showMessageDialog(null, "Your grade is D.");
     else if (testScore < 80)
      JOptionPane.showMessageDialog(null, "Your grade is C.");
     else if (testScore < 90)
      JOptionPane.showMessageDialog(null, "Your grade is B.");
     else
      JOptionPane.showMessageDialog(null, "Your grade is A.");
```

The first `if` statement handled all of the grades less than 60, so when this `if` statement executes, `testScore` will have a value of 60 or greater. If `testScore` is less than 70, the message `"Your grade is D."` is displayed and the rest of the `if-else-if` statement is skipped. This chain of events continues until one of the expressions is found to be true, or the last `else` clause at the end of the statement is encountered.

Notice the alignment and indentation that are used with the `if-else-if` statement: The starting `if` clause, the `else if` clauses, and the trailing `else` clause are all aligned, and the conditionally executed statements are indented.

### Using the Trailing `else` to Catch Errors

The trailing `else` clause, which appears at the end of the `if-else-if` statement, is optional, but in many situations you will use it to catch errors. For example, Code Listing 3-5 will assign the grade `'A'` to any test score that is 90 or greater. What if the highest possible test score is 100? We can modify the code as shown in Code Listing 3-6 so the trailing `else` clause catches any value greater than 100 and displays an error message. Figure 3-13 shows what happens when the user enters a test score that is greater than 100.

**Code Listing 3-6** `(TrailingElse.java)`

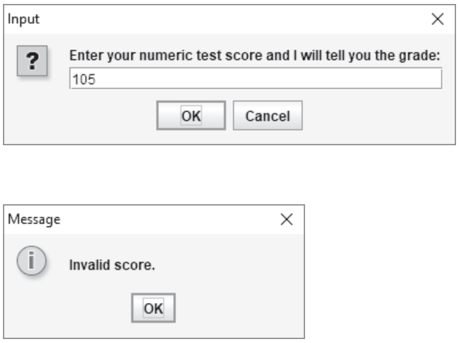
```
import javax.swing.JOptionPane; // Needed for JOptionPane

 /**
     This program asks the user to enter a numeric test
     score and displays a letter grade for the score. The
     program displays an error message if an invalid
     numeric score is entered.
 */

```

```
public class TrailingElse
{
   public static void main(String[] args)
   {
     int testScore;      // Numeric test score
     String input;       // To hold the user's input

     // Get the numeric test score.
     input = JOptionPane.showInputDialog("Enter your numeric " +
                    "test score and I will tell you the grade: ");
     testScore = Integer.parseInt(input);

     // Display the grade.
     if (testScore < 60)
        JOptionPane.showMessageDialog(null, "Your grade is F.");
     else if (testScore < 70)
        JOptionPane.showMessageDialog(null, "Your grade is D.");
     else if (testScore < 80)
        JOptionPane.showMessageDialog(null, "Your grade is C.");
     else if (testScore < 90)
        JOptionPane.showMessageDialog(null, "Your grade is B.");
     else if (testScore <= 100)
        JOptionPane.showMessageDialog(null, "Your grade is A.");
     else
        JOptionPane.showMessageDialog(null, "Invalid score.");

     System.exit(0);
   }
}
```

**Figure 3-13** Interaction with the `NestedDecision` program (Oracle Corporate Counsel)

This input dialog box appears first. The user enters 105 and then clicks the OK button.

This message dialog box appears next.



Enter your numeric test score and I will tell you the grade:

105

Invalid score.

### The `if-else-if` Statement Compared to a Nested Decision Structure

You never have to use the `if-else-if` statement because its logic can be coded with nested `if-else` statements. However, a long series of nested `if-else` statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.

- Because indenting is important in nested statements, a long series of nested `if-else` statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to “wrap around” when printed on paper, making the code even more difficult to read.

The logic of an `if-else-if` statement is usually easier to follow than that of a long series of nested `if-else` statements. And, because all of the clauses are aligned in an `if-else-if` statement, the lengths of the lines in the statement tend to be shorter.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.14 What will the following program display?

```
public class CheckPoint
{
   public static void main(String[] args)
   {
      int funny = 7, serious = 15;
      funny = serious % 2;
      if (funny != 1)
      {
         funny = 0;
         serious = 0;
      }
      else if (funny == 2)
      {
         funny = 10;
         serious = 10;
      }
      else
      {
         funny = 1;
         serious = 1;
      }
      System.out.println(funny + " " + serious);
   }
}
```

3.15 The following program is used in a bookstore to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```
import javax.swing.JOptionPane;
public class CheckPoint
{
   public static void main(String[] args)
   {
      int books, coupons;
      String input;
      input = JOptionPane.showInputDialog("How many books " +
                                    "are being purchased? ");
      books = Integer.parseInt(input);
      if (books < 1)
         coupons = 0;
      else if (books < 3)
         coupons = 1;
      else if (books < 5)
         coupons = 2;
      else
         coupons = 3;
      JOptionPane.showMessageDialog(null,
                  "The number of coupons to give is " +
                  coupons);
      System.exit(0);
   }
}
```

| If the customer purchases this many books . . . | this many coupons are given. |
|---|---|
| 1 | _____ |
| 2 | _____ |
| 3 | _____ |
| 4 | _____ |
| 5 | _____ |
| 10 | _____ |

## 3.5 Logical Operators

**CONCEPT: Logical operators connect two or more relational expressions into one or reverse the logic of an expression.**

Java provides two binary logical operators, `&&` and `||`, which are used to combine two `boolean` expressions into a single expression. It also provides the unary `!` operator, which reverses the truth of a `boolean` expression. Table 3-4 describes these logical operators.

**Table 3-4** Logical operators

| Operator | Meaning | Effect |
|---|---|---|
| `&&` | AND | Connects two `boolean` expressions into one. Both expressions must be `true` for the overall expression to be `true`. |
| `\|\|` | OR | Connects two `boolean` expressions into one. One or both expressions must be `true` for the overall expression to be `true`. It is only necessary for one to be `true`, and it does not matter which one. |
| `!` | NOT | The `!` operator reverses the truth of a `boolean` expression. If it is applied to an expression that is `true`, the operator returns `false`. If it is applied to an expression that is `false`, the operator returns `true`. |

Table 3-5 shows examples of several `boolean` expressions that use logical operators.

**Table 3-5** `boolean` expressions using logical operators

| Expression | Meaning |
|---|---|
| `x > y && a < b` | Is `x` greater than `y` AND is `a` less than `b`? |
| `x == y \|\| x == z` | Is `x` equal to `y` OR is `x` equal to `z`? |
| `!(x > y)` | Is the expression `x > y` NOT `true`? |

Let's take a close look at each of these operators.

### The && Operator

The `&&` operator is known as the logical AND operator. It takes two `boolean` expressions as operands and creates a `boolean` expression that is `true` only when both subexpressions are `true`. Here is an example of an `if` statement that uses the `&&` operator:

```
if (temperature < 20 && minutes > 12)
{
      System.out.println("The temperature is in the " +
                         "danger zone.");
}
```

In this statement the two `boolean` expressions `temperature < 20` and `minutes > 12` are combined into a single expression. The message will be displayed only if `temperature` is less than 20 AND `minutes` is greater than 12. If either `boolean` expression is `false`, the entire expression is `false` and the message is not displayed.

Table 3-6 shows a truth table for the `&&` operator. The truth table lists expressions showing all the possible combinations of `true` and `false` connected with the `&&` operator. The resulting values of the expressions are also shown.

**Table 3-6** Truth table for the && operator

| Expression | Value of the Expression |
|---|---|
| `true && false` | `false` |
| `false && true` | `false` |
| `false && false` | `false` |
| `true && true` | `true` |

As the table shows, both sides of the `&&` operator must be `true` for the operator to return a `true` value.

The `&&` operator performs *short-circuit evaluation*. Here's how it works: If the expression on the left side of the `&&` operator is `false`, the expression on the right side will not be checked. Because the entire expression is `false` if only one of the subexpressions is `false`, it would waste CPU time to check the remaining expression. So, when the `&&` operator finds that the expression on its left is `false`, it short-circuits and does not evaluate the expression on its right.

The `&&` operator can be used to simplify programs that otherwise would use nested `if` statements. The program in Code Listing 3-7 is a different version of the `LoanQualifier` program in Code Listing 3-3, written to use the `&&` operator.

**Code Listing 3-7** `(LogicalAnd.java)`

```java
import java.util.Scanner;

/**
  This program demonstrates the logical && operator.
*/

public class LogicalAnd
{
   public static void main(String[] args)
   {
      double salary;       // Annual salary
      double yearsOnJob;   // Years at current job

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the user's annual salary.
      System.out.print("Enter your annual salary: ");
      salary = keyboard.nextDouble();

      // Get the number of years at the current job.
      System.out.print("Enter the number of years " +
                       "at your current job: ");
      yearsOnJob = keyboard.nextDouble();

      // Determine whether the user qualifies for the loan.
      if (salary >= 50000 && yearsOnJob >= 2)
      {
         System.out.println("You qualify for the loan.");
      }
      else
      {
         System.out.println("You do not qualify.");
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 55000.00 [Enter]
Enter the number of years at your current job: 1 [Enter]
You do not qualify.
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 55000.00 [Enter]
Enter the number of years at your current job: 4 [Enter]
You qualify for the loan.
```

The message `"You qualify for the loan."` is displayed only when both the expressions `salary >= 30000` and `yearsOnJob >= 2` are `true`. If either of these expressions is `false`, the message `"You do not qualify for the loan."` is displayed.

You can also use logical operators with `boolean` variables. For example, assuming that `isValid` is a `boolean` variable, the following `if` statement determines whether `isValid` is `true` and `x` is greater than 90.

```
if (isValid && x > 90)
```

### The || Operator

The `||` operator is known as the logical OR operator. It takes two `boolean` expressions as operands and creates a `boolean` expression that is `true` when either of the subexpressions is `true`. Here is an example of an `if` statement that uses the `||` operator:

```
if (temperature < 20 || temperature > 100)
{
      System.out.println("The temperature is in the " +
                         "danger zone.");
}
```

The message will be displayed if `temperature` is less than 20 OR `temperature` is greater than 100. If either relational test is `true`, the entire expression is `true`.

Table 3-7 shows a truth table for the `||` operator.

All it takes for an OR expression to be `true` is for one side of the `||` operator to be `true`. It doesn't matter if the other side is `false` or `true`. Like the `&&` operator, the `||` operator performs short-circuit evaluation. If the expression on the left side of the `||` operator is `true`, the expression on the right side will not be checked. Because it is necessary for only one of the expressions to be `true`, it would waste CPU time to check the remaining expression.

**Table 3-7** Truth table for the `||` operator

| Expression | Value |
|---|---|
| `true \|\| false` | `true` |
| `false \|\| true` | `true` |
| `false \|\| false` | `false` |
| `true \|\| true` | `true` |

The program in Code Listing 3-8 is a different version of the previous program, shown in Code Listing 3-7. This version uses the `||` operator to determine whether `salary >= 30000` is `true` OR `yearsOnJob >= 2` is `true`. If either expression is `true`, then the person qualifies for the loan.

**Code Listing 3-8** (LogicalOr.java)

```java
import java.util.Scanner;

 /**
   This program demonstrates the logical || operator.
 */

 public class LogicalOr
 {
    public static void main(String[] args)
    {
       double salary;       // Annual salary
       double yearsOnJob;   // Years at current job

       // Create a Scanner object for keyboard input.
       Scanner keyboard = new Scanner(System.in);

       // Get the user's annual salary.
       System.out.print("Enter your annual salary: ");
       salary = keyboard.nextDouble();

       // Get the number of years at the current job.
       System.out.print("Enter the number of years " +
                        "at your current job: ");
       yearsOnJob = keyboard.nextDouble();

       // Determine whether the user qualifies for the loan.
       if (salary >= 50000 || yearsOnJob >= 2)
       {
           System.out.println("You qualify for the loan.");
       }
       else
       {
           System.out.println("You do not qualify.");
       }
    }
 }
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 20000.00 [Enter]
Enter the number of years at your current job: 7 [Enter]
You qualify for the loan.
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 55000.00 [Enter]
Enter the number of years at your current job: 1 [Enter]
You qualify for the loan.
```

**Program Output with Example Input Shown in Bold**

```
Enter your annual salary: 20000.00 [Enter]
Enter the number of years at your current job: 1 [Enter]
You do not qualify.
```

### The ! Operator

The `!` operator performs a logical NOT operation. It is a unary operator that takes a `boolean` expression as its operand and reverses its logical value. In other words, if the expression is `true`, the `!` operator returns `false`, and if the expression is `false`, it returns `true`. Here is an `if` statement using the `!` operator:

```
if (!(temperature > 100))
   System.out.println("This is below the maximum temperature.");
```

First, the expression `(temperature > 100)` is tested and a value of either `true` or `false` is the result. Then the `!` operator is applied to that value. If the expression `(temperature > 100)` is `true`, the `!` operator returns `false`. If the expression `(temperature > 100)` is `false`, the `!` operator returns `true`. The previous code is equivalent to asking: "Is the temperature not greater than 100?"

Table 3-8 shows a truth table for the `!` operator.

**Table 3-8** Truth table for the `!` operator

| Expression | Value |
|---|---|
| `!true` | `false` |
| `!false` | `true` |

## The Precedence of Logical Operators

Like other operators, the logical operators have orders of precedence and associativity. Table 3-9 shows the precedence of the logical operators, from highest to lowest.

**Table 3-9** Logical operators in order of precedence

| |
|---|
| `!` |
| `&&` |
| `\|\|` |

The `!` operator has a higher precedence than many of Java's other operators. You should always enclose its operand in parentheses unless you intend to apply it to a variable or a simple expression with no other operators. For example, consider the following expressions (assume `x` is an `int` variable with a value stored in it):

```
!(x > 2)
!x > 2
```

The first expression applies the `!` operator to the expression `x > 2`. It is asking "is `x` not greater than 2?" The second expression, however, attempts to apply the `!` operator to `x` only.

It is asking "is the logical complement of `x` greater than 2?" Because the `!` operator can only be applied to `boolean` expressions, this statement would cause a compiler error.

The `&&` and `||` operators rank lower in precedence than the relational operators, so precedence problems are less likely to occur. If you are unsure, however, it doesn't hurt to use parentheses anyway.

`(a > b) && (x < y)` *is the same as* `a > b && x < y`
`(x == y) || (b > a)` *is the same as* `x == y || b > a`

The logical operators evaluate their expressions from left to right. In the following expression, `a < b` is evaluated before `y == z`.

```
a < b || y == z
```

In the following expression, `y == z` is evaluated first, however, because the `&&` operator has higher precedence than `||`.

```
a < b || y == z && m > j
```

This expression is equivalent to the following:

```
(a < b) || ((y == z) && (m > j))
```

Table 3-10 shows the precedence of all the operators we have discussed so far. This table includes the assignment, arithmetic, relational, and logical operators.

**Table 3-10** Precedence of all operators discussed so far

| Order of Precedence | Operators | Description |
|---|---|---|
| 1 | `-` (unary negation) `!` | Unary negation, logical NOT |
| 2 | `* / %` | Multiplication, division, modulus |
| 3 | `+ -` | Addition, subtraction |
| 4 | `< > <= >=` | Less than, greater than, less than or equal to, greater than or equal to |
| 5 | `== !=` | Equal to, not equal to |
| 6 | `&&` | Logical AND |
| 7 | `\|\|` | Logical OR |
| 8 | `= += -= *= /= %=` | Assignment and combined assignment |

## Checking Numeric Ranges with Logical Operators

Sometimes you will need to write code that determines whether a numeric value is within a specific range of values or outside a specific range of values. When determining whether a number is inside a range, it's best to use the `&&` operator. For example, the following `if` statement checks the value in `x` to determine whether it is in the range of 20 through 40:

```
if (x >= 20 && x <= 40)
   System.out.println(x + " is in the acceptable range.");
```

The `boolean` expression in the `if` statement will be `true` only when `x` is greater than or equal to 20 AND less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be `true`.

When determining whether a number is outside a range, it's best to use the `||` operator. The following statement determines whether `x` is outside the range of 20 through 40:

```
if (x < 20 || x > 40)
   System.out.println(x + " is outside the acceptable range.");
```

It's important not to get the logic of these logical operators confused. For example, the `boolean` expression in the following `if` statement would never test `true`:

```
if (x < 20 && x > 40)
   System.out.println(x + " is outside the acceptable range.");
```

Obviously, `x` cannot be less than 20 and at the same time be greater than 40.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.16 The following truth table shows various combinations of the values `true` and `false` connected by a logical operator. Complete the table by circling `T` or `F` to indicate whether the result of such a combination is `true` or `false`.

| *Logical Expression* | *Result* (`true` *or* `false`) |
|---|---|
| `true && false` | T F |
| `true && true` | T F |
| `false && true` | T F |
| `false && false` | T F |
| `true \|\| false` | T F |
| `true \|\| true` | T F |
| `false \|\| true` | T F |
| `false \|\| false` | T F |
| `!true` | T F |
| `!false` | T F |

3.17 Assume the variables `a` = 2, `b` = 4, and `c` = 6. Circle the `T` or `F` for each of the following conditions to indicate whether it is `true` or `false`.

| | |
|---|---|
| `a == 4 \|\| b > 2` | T F |
| `6 <= c && a > 3` | T F |
| `1 != b && c != 3` | T F |
| `a >= -1 \|\| a <= b` | T F |
| `!(a > 2)` | T F |

3.18 Write an `if` statement that displays the message `"The number is valid"` if the variable `speed` is within the range 0 through 200.

3.19 Write an `if` statement that displays the message `"The number is not valid"` if the variable `speed` is outside the range 0 through 200.

## 3.6 Comparing String Objects

**CONCEPT: You cannot use relational operators to compare `String` objects. Instead you must use a `String` method.**

You saw in the preceding sections how numeric values can be compared using the relational operators. You should not use the relational operators to compare `String` objects, however. Remember that a `String` object is referenced by a variable that contains the object's memory address. When you use a relational operator with the reference variable, the operator works on the memory address that the variable contains, not the contents of the `String` object. For example, suppose a program has the following declarations:

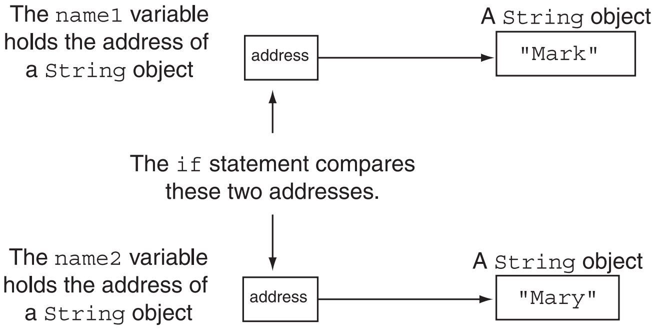
```
String name1 = "Mark";
String name2 = "Mary";
```

And later, the same program has the following `if` statement:

```
if (name1 == name2)
```

The expression `name1 == name2` will be `false`, but not because the strings "Mark" and "Mary" are different. The expression will be `false` because the variables `name1` and `name2` reference different objects. Figure 3-14 illustrates how the variables reference the `String` objects.

**Figure 3-14** The `name1` and `name2` variables reference different `String` objects


The name1 variable holds the address of a String object
address
A String object
"Mark"
The if statement compares these two addresses.
The name2 variable holds the address of a String object
address
A String object
"Mary"

To compare the contents of two `String` objects correctly, you should use the `String` class's `equals` method. The general form of the method is as follows:

```
StringReference1.equals(StringReference2)
```

*`StringReference1`* is a variable that references a `String` object, and *`StringReference2`* is another variable that references a `String` object. The method returns `true` if the two strings are equal, or `false` if they are not equal. Here is an example:

```
if (name1.equals(name2))
```

Assuming that `name1` and `name2` reference `String` objects, the expression in the `if` statement will return `true` if they are the same, or `false` if they are not the same. The program in Code Listing 3-9 demonstrates.

**Code Listing 3-9** (`StringCompare.java`)

```
/**
   This program correctly compares two String objects using
   the equals method.
*/

public class StringCompare
{
   public static void main(String[] args)
   {
      String name1 = "Mark",
             name2 = "Mark",
             name3 = "Mary";

      // Compare "Mark" and "Mark"

      if (name1.equals(name2))
      {
         System.out.println(name1 + " and " + name2 +
                            " are the same.");
      }
      else
      {
         System.out.println(name1 + " and " + name2 +
                            " are NOT the same.");
      }

      // Compare "Mark" and "Mary"

      if (name1.equals(name3))
      {
          System.out.println(name1 + " and " + name3 +
                             " are the same.");
      }
      else
      {
         System.out.println(name1 + " and " + name3 +
                            " are NOT the same.");
      }
   }
}
```

**Program Output**

```
Mark and Mark are the same.
Mark and Mary are NOT the same.
```

You can also compare `String` objects to string literals. Simply pass the string literal as the argument to the `equals` method, as follows:

```
if (name1.equals("Mark"))
```

To determine whether two strings are not equal, simply apply the `!` operator to the `equals` method's return value. Here is an example:

```
if (!name1.equals("Mark"))
```

The `boolean` expression in this `if` statement performs a not-equal-to operation. It determines whether the object referenced by `name1` is not equal to "Mark".

The `String` class also provides the `compareTo` method, which can be used to determine whether one string is greater than, equal to, or less than another string. The general form of the method is as follows:

```
StringReference.compareTo(OtherString)
```

*`StringReference`* is a variable that references a `String` object, and *`OtherString`* is either another variable that references a `String` object or a string literal. The method returns an integer value that can be used in the following manner:

- If the method's return value is negative, the string referenced by *`StringReference`* (the calling object) is less than the *`OtherString`* argument.
- If the method's return value is 0, the two strings are equal.
- If the method's return value is positive, the string referenced by *`StringReference`* (the calling object) is greater than the *`OtherString`* argument.

For example, assume that `name1` and `name2` are variables that reference `String` objects. The following `if` statement uses the `compareTo` method to compare the strings:

```
if (name1.compareTo(name2) == 0)
  System.out.println("The names are the same.");
```

Also, the following expression compares the string referenced by `name1` to the string literal "Joe":

```
if (name1.compareTo("Joe") == 0)
  System.out.println("The names are the same.");
```

The program in Code Listing 3-10 more fully demonstrates the `compareTo` method.

**Code Listing 3-10** **(StringCompareTo.java)**

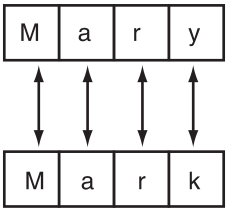
```
/**
   This program compares two String objects using
   the compareTo method.
*/

public class StringCompareTo
{
   public static void main(String[] args)
   {
      String name1 = "Mary",
             name2 = "Mark";

      // Compare "Mary" and "Mark"

      if (name1.compareTo(name2) < 0)
      {
         System.out.println(name1 + " is less than " + name2);
      }
      else if (name1.compareTo(name2) == 0)
      {
         System.out.println(name1 + " is equal to " + name2);
      }
      else if (name1.compareTo(name2) > 0)
      {
         System.out.println(name1 + " is greater than " + name2);
      }
   }
}
```

**Program Output**

```
Mary is greater than Mark
```

Let's take a closer look at this program. When you use the `compareTo` method to compare two strings, the strings are compared character by character. This is often called a *lexicographical comparison*. The program uses the `compareTo` method to compare the strings "Mary" and "Mark", beginning with the first, or leftmost, characters. This is illustrated in Figure 3-15.

**Figure 3-15** String comparison of "Mary" and "Mark"



Here is how the comparison takes place:

1. The "M" in "Mary" is compared with the "M" in "Mark." Because these are the same, the next characters are compared.
2. The "a" in "Mary" is compared with the "a" in "Mark." Because these are the same, the next characters are compared.

3. The “r” in “Mary” is compared with the “r” in “Mark.” Because these are the same, the next characters are compared.
4. The “y” in “Mary” is compared with the “k” in “Mark.” Because these are not the same, the two strings are not equal. The character “y” is greater than “k”, so it is determined that “Mary” is greater than “Mark.”

If one of the strings in a comparison is shorter than the other, Java can only compare the corresponding characters. If the corresponding characters are identical, then the shorter string is considered less than the longer string. For example, suppose the strings “High” and “Hi” were being compared. The string “Hi” would be considered less than “High” because it is shorter.

## Ignoring Case in String Comparisons

The `equals` and `compareTo` methods perform case-sensitive comparisons, which means that uppercase letters are not considered the same as their lowercase counterparts. In other words, “A” is not the same as “a”. This can obviously lead to problems when you want to perform case-insensitive comparisons.

The `String` class provides the `equalsIgnoreCase` and `compareToIgnoreCase` methods. These methods work like the `equals` and `compareTo` methods, except the case of the characters in the strings is ignored. For example, the program in Code Listing 3-11 asks the user to enter the secret word, which is similar to a password. The secret word is `"PROSPERO"`, and the program performs a case-insensitive string comparison to determine whether the user has entered it.

**Code Listing 3-11** `(SecretWord.java)`

```
import java.util.Scanner;

/**
   This program demonstrates a case insensitive string comparison.
*/

public class SecretWord
{
   public static void main(String[] args)
   {
      String input;     // To hold the user's input

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Prompt the user to enter the secret word.
      System.out.print("Enter the secret word: ");
      input = keyboard.nextLine();

      // Determine whether the user entered the secret word.
      if (input.equalsIgnoreCase("PROSPERO"))
```

```
    {
       System.out.println("Congratulations! You know the " +
                          "secret word!");
    }
    else
    {
        System.out.println("Sorry, that is NOT the " +
                           "secret word!");
    }
  }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the secret word: Ferdinand [Enter]
Sorry, that is NOT the secret word!
```

**Program Output with Example Input Shown in Bold**

```
Enter the secret word: Prospero [Enter]
Congratulations! You know the secret word!
```

The `compareToIgnoreCase` method works exactly like the `compareTo` method, except the case of the characters in the strings being compared is ignored.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

3.20 Assume the variable `name` references a `String` object. Write an `if` statement that displays "Do I know you?" if the `String` object contains "Timothy".

3.21 Assume the variables `name1` and `name2` reference two different `String` objects, containing different strings. Write code that displays the strings referenced by these variables in alphabetical order.

3.22 Modify the statement you wrote in response to Checkpoint 3.20 so it performs a case-insensitive comparison.

## 3.7 More about Variable Declaration and Scope

**CONCEPT:** The scope of a variable is limited to the block in which it is declared.

Recall from Chapter 2 that a local variable is a variable that is declared inside a method. Java allows you to create local variables just about anywhere in a method. For example, look at the program in Code Listing 3-12. The `main` method declares two `String` reference variables: `firstName` and `lastName`. Notice that the declarations of these variables appear near the code that first uses the variables.

**Code Listing 3-12** `(VariableScope.java)`

```java
 1 import java.util.Scanner;
 2
 3 /**
 4    This program demonstrates how variables may be declared
 5    in various locations throughout a program.
 6 */
 7
 8 public class VariableScope
 9 {
10    public static void main(String[] args)
11    {
12       // Create a Scanner object for keyboard input.
13       Scanner keyboard = new Scanner(System.in);
14
15       // Get the user's first name.
16       System.out.print("Enter your first name: ");
17       String firstName;
18       firstName = keyboard.nextLine();
19
20       // Get the user's last name.
21       System.out.print("Enter your last name: ");
22       String lastName;
23       lastName = keyboard.nextLine();
24
25       // Display a message.
26       System.out.println("Hello " + firstName + " " + lastName);
27    }
28 }
```

**Program Output with Example Input Shown in Bold**

```
Enter your first name: Amy [Enter]
Enter your last name: Jones [Enter]
Hello Amy Jones
```

Although it is a common practice to declare all of a method's local variables at the beginning of the method, it is possible to declare them at later points. Sometimes programmers declare certain variables near the part of the program where they are used in order to make their purpose more evident.

Recall from Chapter 2 that a variable's scope is the part of the program where the variable's name may be used. A local variable's scope always starts at the variable's declaration, and ends at the closing brace of the block of code in which it is declared. In Code Listing 3-12, the `firstName` variable is visible only to the code in lines 18 through 26. The `lastName` variable is visible only to the code in lines 23 through 26.

**NOTE:** When a program is running and it enters the section of code that constitutes a variable's scope, it is said that the variable "comes into scope." This simply means the variable is now visible and the program may reference it. Likewise, when a variable "leaves scope" it may not be used.

## 3.8

## The Conditional Operator (Optional)

**CONCEPT: You can use the conditional operator to create short expressions that work like `if-else` statements.**

The *conditional operator* is powerful and unique. Because it takes three operands, it is considered a ternary operator. The conditional operator provides a shorthand method of expressing a simple `if-else` statement. The operator consists of the question mark (`?`) and the colon (`:`). You use the operator to write a conditional expression, in the following format:

```
BooleanExpression ? Value1: Value2;
```

The *`BooleanExpression`* is like the `boolean` expression in the parentheses of an `if` statement. If the *`BooleanExpression`* is true, then the value of the conditional expression is *`Value1`*. Otherwise, the value of the conditional expression is *`Value2`*. Here is an example of a statement using the conditional operator:

```
y = x < 0 ? 10: 20;
```

This preceding statement performs the same operation as the following `if-else` statement:

```
if (x < 0)
    y = 10;
else
    y = 20;
```

The conditional operator gives you the ability to pack decision-making power into a concise line of code. With a little imagination it can be applied to many other programming problems. For instance, consider the following statement:

```
System.out.println("Your grade is: " +
            (score < 60 ? "Fail." : "Pass."));
```

Converted to an `if-else` statement, it would be written as follows:

```
if (score < 60)
    System.out.println("Your grade is: Fail.");
else
    System.out.println("Your grade is: Pass.");
```

**NOTE:** The parentheses are placed around the conditional expression because the `+` operator has higher precedence than the `?:` operator. Without the parentheses, the `+` operator would concatenate the value in score with the string `"Your grade is: "`.

For a complete example using the conditional operator, see the program named *ConsultantCharges.java* in this chapter's source code folder, available for download from the book's companion Web site (www.pearsonhighered.com/gaddis).

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.23 Rewrite the following `if-else` statements as statements that use the conditional operator.

a)
```
if (x > y)
    z = 1;
else
    z = 20;
```

b)
```
if (temp > 45)
    population = base * 10;
else
    population = base * 2;
```

c)
```
if (hours > 40)
    wages *= 1.5;
else
    wages *= 1;
```

d)
```
if (result >= 0)
    System.out.println("The result is positive.");
else
    System.out.println("The result is negative.");
```

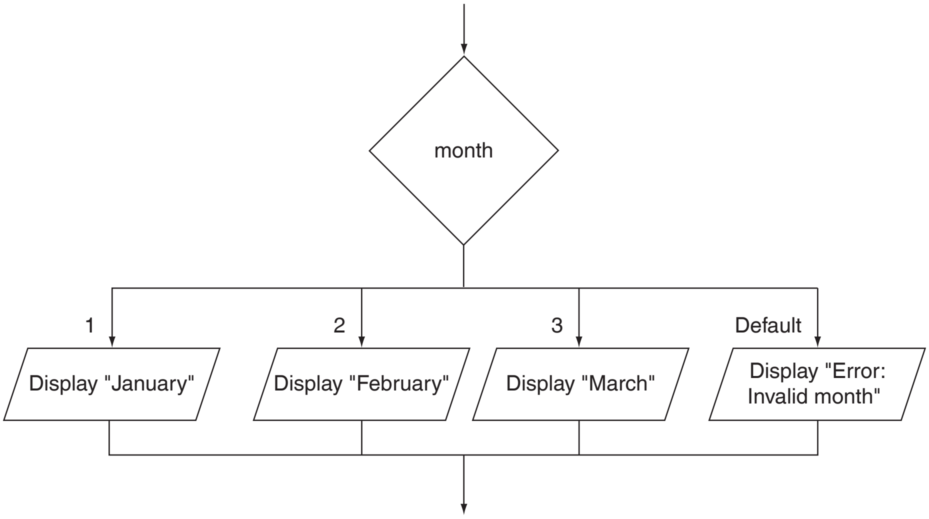
## 3.9 The switch Statement

**CONCEPT: The switch statement lets the value of a variable or expression determine where the program will branch to.**

The *`switch` statement* is a *multiple alternative decision structure*. It allows you to test the value of a variable or an expression and then use that value to determine which statement or set of statements to execute. Figure 3-16 shows an example of how a multiple alternative decision structure looks in a flowchart.

In the flowchart, the diamond symbol shows `month`, which is the name of a variable. If the `month` variable contains the value 1, the program displays *January*. If the `month` variable contains the value 2, the program displays *February*. If the `month` variable contains the value 3, the program displays *March*. If the `month` variable contains none of these values, the action that is labeled ***Default*** is executed. In this case, the program displays *Error: Invalid month*.

**Figure 3-16** A multiple alternative decision structure


month
1
2
3
Default
Display "January"
Display "February"
Display "March"
Display "Error: Invalid month"

Here is the general format of a `switch` statement in Java:

*The* `testExpression` *is a variable or expression.*

```
switch (testExpression)
{
   case value_1:
      statement;
      statement;
      etc.
      break;
```

*These statements are executed if the* `testExpression` *is equal to* `value_1`*.*

```
   case value_2:
      statement;
      statement;
      etc.
      break;
```

*These statements are executed if the* `testExpression` *is equal to* `value_2`*.*

*Insert as many* `case` *sections as necessary.*

```
   case value_N:
      statement;
      statement;
      etc.
      break;
```

*These statements are executed if the* `testExpression` *is equal to* `value_N`*.*

```
   default:
      statement;
      statement;
      etc.
      break;
}
```

*These statements are executed if the* `testExpression` *is not equal to any of the* `case` *values.*

The first line of the statement starts with the word `switch`, followed by a *`testExpression`*, which is enclosed in parentheses. The *`testExpression`* is a variable or an expression that gives a `char`, `byte`, `short`, `int`, or string value.

Beginning at the next line is a block of code enclosed in curly braces. Inside this block of code is one or more `case` sections. A `case` section begins with the word `case`, followed by a value, followed by a colon. Each `case` section contains one or more statements, followed by a `break` statement. After all of the `case` sections, an optional `default` section appears.

When the `switch` statement executes, it compares the value of the *`testExpression`* with the values that follow each of the `case` statements (from top to bottom). When it finds a `case` value that matches the *`testExpression`*'s value, the program branches to the `case` statement. The statements that follow the `case` statement are executed until a `break` statement is encountered. At that point, the program jumps out of the `switch` statement. If the *`testExpression`* does not match any of the `case` values, the program branches to the `default` statement and executes the statements that immediately follow it.

**NOTE:** Each of the `case` values must be unique.

For example, the following code performs the same operation as the flowchart in Figure 3-16. Assume `month` is an `int` variable.

```
switch (month)
{
    case 1:
       System.out.println("January");
       break;
    case 2:
       System.out.println("February");
       break;
    case 3:
       System.out.println("March");
       break;
    default:
      System.out.println("Error: Invalid month");
      break;
}
```

In this example the *`testExpression`* is the `month` variable. The `month` variable will be evaluated and one of the following actions will take place:

- If the value in the `month` variable is 1, the program will branch to the `case 1:` section and execute the `System.out.println("January")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is 2, the program will branch to the `case 2:` section and execute the `System.out.println("February")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.

- If the value in the `month` variable is 3, the program will branch to the `case 3:` section and execute the `System.out.println("March")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is not 1, 2, or 3, the program will branch to the `default:` section and execute the `System.out.println("Error: Invalid month")` statement that immediately follows it.

The `switch` statement can be used as an alternative to an `if-else-if` statement that compares the same variable or expression to several different values. For example, the previously shown `switch` statement works like this `if-else-if` statement:

```
if (month == 1)
{
    System.out.println("January");
}
else if (month == 2)
{
    System.out.println("February");
}
else if (month == 3)
{
    System.out.println("March");
}
else
{
    System.out.println("Error: Invalid month");
}
```

**NOTE:** The `default` section is optional. If you leave it out, however, the program will have nowhere to branch to if the *`testExpression`* doesn't match any of the case values.

The program in Code Listing 3-13 shows how a simple `switch` statement works.

**Code Listing 3-13** **(`SwitchDemo.java`)**

```
import java.util.Scanner;

/**
   This program demonstrates the switch statement.
*/

public class SwitchDemo
{
    public static void main(String[] args)
    {
      int number;    // A number entered by the user

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get one of the numbers 1, 2, or 3 from the user.
      System.out.print("Enter 1, 2, or 3: ");
      number = keyboard.nextInt();

      // Determine the number entered.
      switch (number)
      {
         case 1:
            System.out.println("You entered 1.");
            break;
         case 2:
            System.out.println("You entered 2.");
            break;
         case 3:
            System.out.println("You entered 3.");
            break;
         default:
            System.out.println("That's not 1, 2, or 3!");
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter 1, 2, or 3: 2 [Enter]
You entered 2.
```

**Program Output with Example Input Shown in Bold**

```
Enter 1, 2, or 3: 5 [Enter]
That's not 1, 2, or 3!
```

Notice the `break` statements that are in the `case 1`, `case 2`, and `case 3` sections.

```
switch (number)
{
   case 1:
         System.out.println("You entered 1.");
          break;←
   case 2:
         System.out.println("You entered 2.");
          break;←
   case 3:
         System.out.println("You entered 3.");
         break;←
   default:
         System.out.println("That's not 1, 2, or 3!");
}
```

The `case` statements show the program where to start executing in the block and the `break` statements show the program where to stop. Without the `break` statements, the program would execute all of the lines from the matching `case` statement to the end of the block.

> **NOTE:** The `default` section (or the last `case` section if there is no `default`) does not need a `break` statement. Some programmers prefer to put one there anyway for consistency.

The program in Code Listing 3-14 is a modification of Code Listing 3-13, without the `break` statements.

**Code Listing 3-14** (`NoBreaks.java`)

```java
import java.util.Scanner;

/**
   This program demonstrates the switch statement.
*/

public class NoBreaks
{
   public static void main(String[] args)
   {
      int number;   // A number entered by the user

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get one of the numbers 1, 2, or 3 from the user.
      System.out.print("Enter 1, 2, or 3: ");
      number = keyboard.nextInt();

      // Determine the number entered.
      switch (number)
      {
         case 1:
            System.out.println("You entered 1.");
         case 2:
            System.out.println("You entered 2.");
         case 3:
            System.out.println("You entered 3.");
         default:
            System.out.println("That's not 1, 2, or 3!");
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter 1, 2, or 3:  1 [Enter]
You entered 1.
You entered 2.
You entered 3.
That's not 1, 2, or 3!
```

**Program Output with Example Input Shown in Bold**

```
Enter 1, 2, or 3:  3 [Enter]
You entered 3.
That's not 1, 2, or 3!
```

Without the `break` statement, the program "falls through" all of the statements below the one with the matching `case` expression. Sometimes this is what you want. For instance, the program in Code Listing 3-15 asks the user to select a grade of pet food. The available choices are A, B, and C. The `switch` statement will recognize either uppercase or lowercase letters.

**Code Listing 3-15** (`PetFood.java`)

```java
import java.util.Scanner;

/**
   This program demonstrates a switch statement.
*/

public class PetFood
{
   public static void main(String[] args)
   {
      String input;   // To hold the user's input
      char foodGrade;  // Grade of pet food

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Prompt the user for a grade of pet food.
      System.out.println("Our pet food is available in " +
                         "three grades:");
      System.out.print("A, B, and C. Which do you want " +
                       "pricing for? ");
      input = keyboard.nextLine();
      foodGrade = input.charAt(0);

      // Display pricing for the selected grade.
      switch(foodGrade)
      {
         case 'a':
         case 'A':
            System.out.println("30 cents per lb.");
            break;
         case 'b':
         case 'B':
            System.out.println("20 cents per lb.");
            break;
         case 'c':
         case 'C':
            System.out.println("15 cents per lb.");
            break;
         default:
            System.out.println("Invalid choice.");
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Our pet food is available in three grades:
A, B, and C. Which do you want pricing for? b [Enter]
20 cents per lb.
```

**Program Output with Example Input Shown in Bold**

```
Our pet food is available in three grades:
A, B, and C. Which do you want pricing for? B [Enter]
20 cents per lb.
```

When the user enters ‘a’ the corresponding `case` has no statements associated with it, so the program falls through to the next `case`, which corresponds with ‘A’.

```
case 'a':
case 'A':
     System.out.println("30 cents per lb.");
     break;
```

The same technique is used for ‘b’ and ‘c’.

If you are using a version of Java prior to Java 7, a `switch` statement’s *`testExpression`* can be a `char`, `byte`, `short`, or `int` value. Beginning with Java 7, however, the *`testExpression`* can also be a string. The program in Code Listing 3-16 demonstrates.

**Code Listing 3-16** (`Seasons.java`)

```java
import java.util.Scanner;

/**
   This program translates the English names of
   the seasons into Spanish.
*/

public class Seasons
{
   public static void main(String[] args)
   {
      String input;

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get a day from the user.
      System.out.print("Enter the name of a season: ");
      input = keyboard.nextLine();

      // Translate the season to Spanish.
      switch (input)
      {
         case "spring":
            System.out.println("la primavera");
            break;
         case "summer":
            System.out.println("el verano");
            break;
         case "autumn":
         case "fall":
            System.out.println("el otono");
            break;
         case "winter":
            System.out.println("el invierno");
            break;
         default:
            System.out.println("Please enter one of these words:\n"+
                         "spring, summer, autumn, fall, or winter.");
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the name of a season: summer [Enter]
el verano
```

**Program Output with Example Input Shown in Bold**

```
Enter the name of a season: fall [Enter]
el otono
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.24 Complete the following program skeleton by writing a `switch` statement that displays “one” if the user has entered 1, “two” if the user has entered 2, and “three” if the user has entered 3. If a number other than 1, 2, or 3 is entered, the program should display an error message.

```
import java.util.Scanner;
public class CheckPoint
{
   public static void main(String[] args)
   {
      int userNum;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter one of the numbers " +
                       "1, 2, or 3: ");
      userNum = keyboard.nextInt();
      //
      // Write the switch statement here.
      //
   }
}
```

3.25 Rewrite the following `if-else-if` statement as a `switch` statement.

```
if (selection == 'A')
    System.out.println("You selected A.");
else if (selection == 'B')
    System.out.println("You selected B.");
else if (selection == 'C')
    System.out.println("You selected C.");
else if (selection == 'D')
    System.out.println("You selected D.");
else
    System.out.println("Not good with letters, eh?");
```

3.26 Explain why you cannot convert the following `if-else-if` statement into a `switch` statement.

```
if (temp == 100)
    x = 0;
else if (population > 1000)
    x = 1;
else if (rate < .1)
    x = -1;
```

3.27 What is wrong with the following `switch` statement?

```
// This code has errors!!!
switch (temp)
{
    case temp < 0 :
       System.out.println("Temp is negative.");
       break;
    case temp = 0:
       System.out.println("Temp is zero.");
       break;
    case temp > 0 :
       System.out.println("Temp is positive.");
       break;
}
```

3.28 What will the following code display?

```
int funny = 7, serious = 15;
funny = serious * 2;
switch (funny)
{ case 0 :
     System.out.println("That is funny.");
     break;
  case 30:
     System.out.println("That is serious.");
     break;
  case 32:
     System.out.println("That is seriously funny.");
     break;
  default:
     System.out.println(funny);
}
```

## 3.10 Displaying Formatted Output with `System.out.printf` and `String.format`

**CONCEPT: The `System.out.printf` method allows you to format output in a variety of ways. The `String.format` method allows you to format a string, without displaying it. The string can be displayed at a later time.**

When you display numbers with the `System.out.println` or `System.out.print` method, you have little control over the way the numbers appear. For example, a value of the `double` data type can be displayed with as many as 15 decimal places, as demonstrated by the following code:

```
double number = 10.0 / 6.0;
System.out.println(number);
```

This code will display:

```
1.666666666666667
```

Quite often, you want to format numbers so they are displayed in a particular way. For example, you might want to round a floating-point number to a specific number of decimal places, or insert comma separators to make a number easier to read. Fortunately, Java gives us a way to do just that, and more, with the `System.out.printf` method. The method's general format is as follows:

```
System.out.printf(FormatString, ArgumentList)
```

In the general format, *`FormatString`* is a string that contains text, special formatting specifiers, or both. *`ArgumentList`* is a list of zero or more additional arguments, which will be formatted according to the format specifiers listed in the format string.

The simplest way you can use the `System.out.printf` method is with only a format string, and no additional arguments. Here is an example:

```
System.out.printf("I love Java programming.\n");
```

The format string in this example is `"I love Java programming.\n"`. This method call does not perform any special formatting, however. It simply prints the string `"I love Java programming.\n"`. Using the method in this fashion is exactly like using the `System.out.print` method.

In most cases you will call the `System.out.printf` method in the following manner:

- The format string will contain one or more format specifiers. A *format specifier* is a placeholder for a value that will be inserted into the string when it is displayed.
- After the format string, one or more additional arguments will appear. Each of the additional arguments will correspond to a format specifier that appears inside the format string.

The following code shows an example:

```
double sales = 12345.67;
System.out.printf("Our sales are %f for the day.\n", sales);
```

Notice the following characteristics of the `System.out.printf` method call:

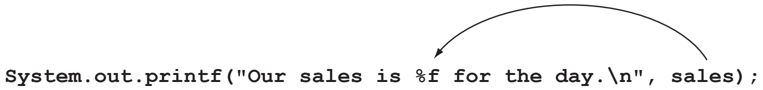
- Inside the format string, the `%f` is a format specifier. The letter `f` indicates that a floating-point value will be inserted into the string when it is displayed.
- Following the format string, the `sales` variable is passed as an argument. This argument corresponds to the `%f` format specifier that appears inside the format string.

When the `System.out.printf` method executes, the `%f` will not be displayed on the screen. In its place, the value of the `sales` argument will be displayed. Here is the output of the code:

```
Our sales are 12345.670000 for the day.
```

The diagram in Figure 3-17 shows how the `sales` variable corresponds to the `%f` format specifier.

**Figure 3-17** The value of the sales variable is displayed in the place of the `%f` format specifier



```
System.out.printf("Our sales is %f for the day.\n", sales);
```

Here is another example:

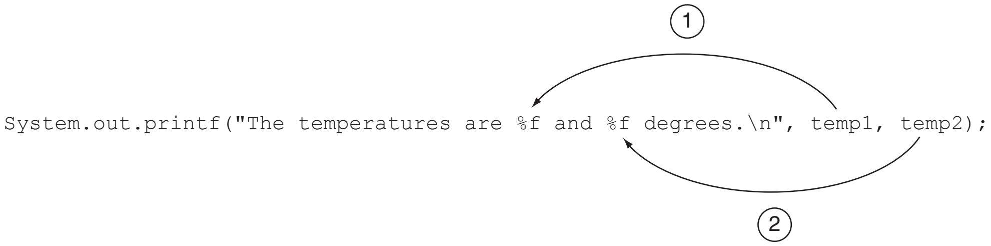
```
double temp1 = 72.5, temp2 = 83.7;
System.out.printf("The temperatures are %f and %f degrees.\n", temp1, temp2);
```

First, notice that this example uses two `%f` format specifiers in the format string. Also notice that two additional arguments appear after the format string. The value of the first argument, `temp1`, will be printed in place of the first `%f`, and the value of the second argument, `temp2`, will be printed in place of the second `%f`. The code will produce the following output:

```
The temperatures are 72.500000 and 83.700000 degrees.
```

There is a one-to-one correspondence between the format specifiers and the arguments that appear after the format string. The diagram in Figure 3-18 shows how the first format specifier corresponds to the first argument after the format string (the `temp1` variable), and the second format specifier corresponds to the second argument after the format string (the `temp2` variable).

**Figure 3-18** The format specifiers and their corresponding arguments



```
System.out.printf("The temperatures are %f and %f degrees.\n", temp1, temp2);
```

The following code shows another example:

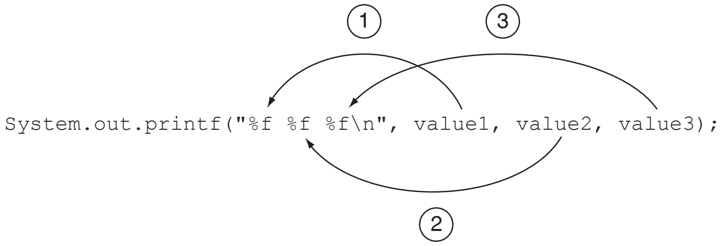
```
double value1 = 3.0;
double value2 = 6.0;
double value3 = 9.0;
System.out.printf("%f %f %f\n", value1, value2, value3);
```

In the `System.out.printf` method call, there are three format specifiers and three additional arguments after the format string. This code will produce the following output:

```
3.000000 6.000000 9.000000
```

The diagram in Figure 3-19 shows how the format specifiers correspond to the arguments that appear after the format string.

**Figure 3-19** The format specifiers and their corresponding arguments



The previous examples demonstrated how to format floating-point numbers with the `%f` format specifier. The letter `f` in the format specifier is a *conversion character* that indicates the data type of the argument that is being formatted. You use the `f` conversion character with any argument that is a `float` or a `double`.

If you want to format an integer value, you must use the `%d` format specifier. The `d` conversion character stands for decimal integer, and it can be used with arguments of the `int`, `short`, and `long` data types. Here is an example that displays an `int`:

```
int hours = 40;
System.out.printf("I worked %d hours this week.\n", hours);
```

In this example, the `%d` format specifier corresponds with the `hours` argument. This code will display the following:

```
I worked 40 hours this week.
```

Here is an example that displays two `int` values:

```
int dogs = 2;
int cats = 4;
System.out.printf("We have %d dogs and %d cats.\n", dogs, cats);
```

This code will display the following:

```
We have 2 dogs and 4 cats.
```

Keep in mind that `%f` must be used with floating-point values, and `%d` must be used with integer values. Otherwise, an error will occur at runtime.

## Format Specifier Syntax

In the previous examples you saw how format specifiers correspond to the arguments that appear after the format string. Now you can learn how to use format specifiers to actually format the values that they correspond to. When displaying numbers, the general syntax for writing a format specifier is:

```
%[flags][width][.precision]conversion
```

The items that appear inside brackets are optional. Here is a summary of each item:

- `%`—All format specifiers begin with a `%` character.
- *`flags`*—After the `%` character, one or more optional flags may appear. Flags cause the value to be formatted in a variety of ways.

- *`width`*—After any flags, you can optionally specify the minimum field width for the value.
- *`.precision`*—If the value is a floating-point number, after the minimum field width, you can optionally specify the precision. This is the number of decimal places that the number should be rounded to.
- *`conversion`*—All format specifiers must end with a conversion character, such as `f` for floating-point, or `d` for decimal integer.

Let's take a closer look at each of the optional items, beginning with precision.

## Precision

You probably noticed in the previous examples that the `%f` format specifier causes floating-point values to be displayed with six decimal places. You can change the number of decimal points that are displayed, as shown in the following example:

```
double temp = 78.42819;
System.out.printf("The temperature is %.2f degrees.\n", temp);
```

Notice that this example doesn't use the regular `%f` format specifier, but uses `%.2f` instead. The `.2` that appears between the `%` and the `f` specifies the *precision* of the displayed value. It will cause the value of the `temp` variable to be rounded to two decimal places. This code will produce the following output:

```
The temperature is 78.43 degrees.
```

The following example displays the same value, rounded to one decimal place:

```
double temp = 78.42819;
System.out.printf("The temperature is %.1f degrees.\n", temp);
```

This code will produce the following output:

```
The temperature is 78.4 degrees.
```

The following code shows another example:

```
double value1 = 123.45678;
double value2 = 123.45678;
double value3 = 123.45678;
System.out.printf("%.1f %.2f %.3f\n", value1, value2, value3);
```

In this example, `value1` is rounded to one decimal place, `value2` is rounded to two decimal places, and `value3` is rounded to three decimal places. This code will produce the following output:

```
123.5 123.46 123.457
```

Keep in mind that you can specify precision only with floating-point point values. If you specify a precision with the `%d` format specifier, an error will occur at runtime.

## Specifying a Minimum Field Width

A format specifier can also include a *minimum field width*, which is the minimum number of spaces that should be used to display the value. The following example prints a floating-point number in a field that is 20 spaces wide:

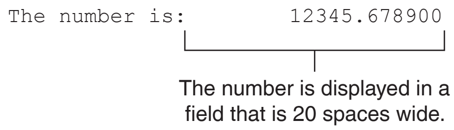
```
double number = 12345.6789;
System.out.printf("The number is:%20f\n", number);
```

Notice that the number 20 appears in the format specifier, between the `%` and the `f`. This code will produce the following output:

```
The number is:    12345.678900
```

In this example, the 20 that appears inside the `%f` format specifier indicates that the number should be displayed in a field that is a minimum of 20 spaces wide. This is illustrated in Figure 3-20.

**Figure 3-20** The number is displayed in a field that is 20 spaces wide


The number is:        12345.678900
The number is displayed in a field that is 20 spaces wide.

In this case, the number that is displayed is shorter than the field in which it is displayed. The number `12345.678900` uses only 12 spaces on the screen, but it is displayed in a field that is 20 spaces wide. When this is the case, the number will be right-justified in the field. If a value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it. The following example prints a floating-point number in a field that is only one space wide:

```
double number = 12345.6789;
System.out.printf("The number is:%1f\n", number);
```

The value of the number variable requires more than one space, however, so the field width is expanded to accommodate the entire number. This code will produce the following output:

```
The number is:12345.678900
```

You can specify a minimum field width for integers, as well as for floating-point values. The following example displays an integer with a minimum field width of six characters:

```
int number = 200;
System.out.printf("The number is:%6d", number);
```

This code will display the following:

```
The number is:  200
```

### Combining Minimum Field Width and Precision in the Same Format Specifier

When specifying the minimum field width and the precision of a floating-point number in the same format specifier, remember that the field width must appear first, followed by the precision. For example, the following code displays a number in a field of 12 spaces, rounded to two decimal places:

```
double number = 12345.6789;
System.out.printf("The number is:%12.2f\n", number);
```

This code will produce the following output:

```
The number is:  12345.68
```

Field widths can help when you need to print numbers aligned in columns. For example, look at Code Listing 3-17. Each of the variables is displayed in a field that is eight spaces wide, and rounded to two decimal places. The numbers appear aligned in a column.

**Code Listing 3-17** `(Columns.java)`

```java
/**
   This program displays a variety of
   floating-point numbers in a column
   with their decimal points aligned.
*/

public class Columns
{
   public static void main(String[] args)
   {
      // Declare a variety of double variables.
      double num1 = 127.899;
      double num2 = 3465.148;
      double num3 = 3.776;
      double num4 = 264.821;
      double num5 = 88.081;
      double num6 = 1799.999;

      // Display each variable in a field of
      // 8 spaces with 2 decimal places.
      System.out.printf("%8.2f\n", num1);
      System.out.printf("%8.2f\n", num2);
      System.out.printf("%8.2f\n", num3);
      System.out.printf("%8.2f\n", num4);
      System.out.printf("%8.2f\n", num5);
      System.out.printf("%8.2f\n", num6);
   }
}
```

**Program Output**

```
  127.90
 3465.15
    3.78
  264.82
   88.08
 1800.00
```

## Flags

There are several optional flags that you can insert into a format specifier to cause a value to be formatted in a particular way. In this book, we will use flags for the following purposes:

- To display numbers with comma separators
- To pad numbers with leading zeros
- To left-justify numbers

If you use a flag in a format specifier, you must write the flag before the field width and the precision.

### Comma Separators

Large numbers are easier to read if they are displayed with comma separators. You can format a number with comma separators by inserting a comma (`,`) flag into the format specifier. Here is an example:

```
double amount = 1234567.89;
System.out.printf("%,f\n", amount);
```

This code will produce the following output:

```
1,234,567.890000
```

Quite often, you will want to format a number with comma separators, and round the number to a specific number of decimal places. You can accomplish this by inserting a comma, followed by the precision value, into the `%f` format specifier, as shown in the

following example:

```
double sales = 28756.89;
System.out.printf("Sales for the month are %,.2f\n", sales);
```

This code will produce the following output:

```
Sales for the month are 28,756.89
```

Code Listing 3-18 demonstrates how the comma separator and a precision of two decimal places can be used to format a number as a currency amount.

**Code Listing 3-18** (`CurrencyFormat.java`)

```
/**
   This program demonstrates how to use the System.out.printf
   method to format a number as currency.
*/

public class CurrencyFormat
{
   public static void main(String[] args)
   {
      double monthlyPay = 5000.0;
      double annualPay = monthlyPay * 12;
      System.out.printf("Your annual pay is $%,.2f\n", annualPay);
   }
}
```

**Program Output**

```
Your annual pay is $60,000.00
```

The following example displays a floating-point number with comma separators, in a field of 15 spaces, rounded to two decimal places:

```
double amount = 1234567.8901;
System.out.printf("%,15.2f\n", amount);
```

This code will produce the following output:

```
1,234,567.89
```

The following example displays an `int` with a minimum field width of six characters:

```
int number = 200;
System.out.printf("The number is:%6d", number);
```

This code will display the following:

```
The number is: 200
```

The following example displays an `int` with comma separators, with a minimum field width of 10 characters:

```
int number = 20000;
System.out.printf("The number is:%,10d", number);
```

This code will display the following:

```
The number is: 20,000
```

### Padding Numbers with Leading Zeros

Sometimes, when a number is shorter than the field in which it is displayed, you want to pad the number with leading zeros. If you insert a `0` flag into a format specifier, the resulting number will be padded with leading zeros, if it is shorter than the field width. The following code shows an example:

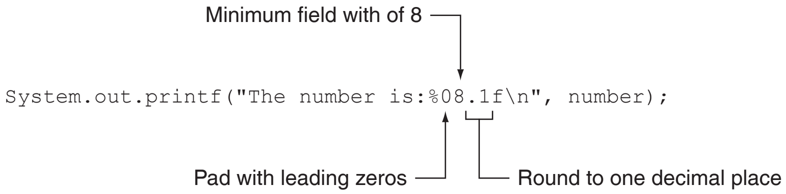
```
double number = 123.4;
System.out.printf("The number is:%08.1f\n", number);
```

This code will produce the following output:

```
The number is:000123.4
```

The diagram in Figure 3-21 shows the purpose of each part of the format specifier in the previous example.

**Figure 3-21** Format specifier that pads with leading zeros



```
System.out.printf("The number is:%08.1f\n", number);
```

The following example displays an `int` padded with leading zeros, with a minimum field width of seven characters:

```
int number = 1234;
System.out.printf("The number is:%07d", number);
```

This code will display the following:

```
The number is:0001234
```

The program in Code Listing 3-19 shows another example. This program displays a variety of floating-point numbers with leading zeros, in a field of nine spaces, rounded to two decimal places.

**Code Listing 3-19** `(LeadingZeros.java)`

```
/**
   This program displays numbers padded with leading zeros.
*/

public class LeadingZeros
{
   public static void main(String[] args)
   {
      // Declare a variety of double variables.
      double number1 = 1.234;
      double number2 = 12.345;
      double number3 = 123.456;

      // Display each variable with leading
      // zeros, in a field of 9 spaces, rounded
      // to 2 decimal places.
      System.out.printf("%09.2f\n", number1);
      System.out.printf("%09.2f\n", number2);
      System.out.printf("%09.2f\n", number3);
   }
}
```

**Program Output**

```
000001.23
000012.35
000123.46
```

### Left-Justifying Numbers

By default, when a number is shorter than the field in which it is displayed, the number is right-justified within that field. If you want a number to be left-justified within its field, you insert a minus sign (`-`) flag into the format specifier. Code Listing 3-20 shows an example.

**Code Listing 3-20** (`LeftJustified.java`)

```
/**
   This program displays a variety of
   numbers left-justified in columns.
*/

public class LeftJustified
{
   public static void main(String[] args)
   {
      // Declare a variety of int variables.
      int num1 = 123;
      int num2 = 12;
      int num3 = 45678;
      int num4 = 456;
      int num5 = 1234567;
      int num6 = 1234;

      // Display each variable left-justified
      // in a field of 8 spaces.
      System.out.printf("%-8d%-8d\n", num1, num2);
      System.out.printf("%-8d%-8d\n", num3, num4);
      System.out.printf("%-8d%-8d\n", num5, num6);
   }
}
```

**Program Output**

```
123     12
45678   456
1234567 1234
```

## Formatting String Arguments

If you wish to print a string argument, use the `%s` format specifier. Here is an example:

```
String name = "Ringo";
System.out.printf("Your name is %s\n", name);
```

This code produces the following output:

```
Your name is Ringo
```

You can also use a field width when printing strings. For example, look at the following code:

```
String name1 = "George";
String name2 = "Franklin";
String name3 = "Jay";
String name4 = "Ozzy";
String name5 = "Carmine";
String name6 = "Dee";
System.out.printf("%10s%10s\n", name1, name2);
System.out.printf("%10s%10s\n", name3, name4);
System.out.printf("%10s%10s\n", name5, name6);
```

The `%10s` format specifier prints a string in a field that is ten spaces wide. This code displays the values of the variables in a table with three rows and two columns. Each column has a width of ten spaces. Here is the output of the code:

```
George Franklin
   Jay     Ozzy
Carmine     Dee
```

Notice that the strings are right-justified. You can use the minus flag (`-`) to left-justify a string within its field. The following code demonstrates:

```
String name1 = "George";
String name2 = "Franklin";
String name3 = "Jay";
String name4 = "Ozzy";
String name5 = "Carmine";
String name6 = "Dee";
System.out.printf("%-10s%-10s\n", name1, name2);
System.out.printf("%-10s%-10s\n", name3, name4);
System.out.printf("%-10s%-10s\n", name5, name6);
```

Here is the output of the code:

```
George  Franklin
Jay     Ozzy
Carmine Dee
```

The following example shows how you can print arguments of different data types:

```
int hours = 40;
double pay = hours * 25;
String name = "Jay";
System.out.printf("Name: %s, Hours: %d, Pay: $%,.2f\n",
                  name, hours, pay);
```

In this example, we are displaying a `String`, an `int`, and a `double`. The code will produce the following output:

```
Name: Jay, Hours: 40, Pay: $1,000.00
```

**NOTE:** The format specifiers we have shown in this section are the basic ones. Java provides much more powerful format specifiers for more complex formatting needs. The API documentation gives an overview of them all.

## The `String.format` Method

The `System.out.printf` method formats a string and displays it in the console window. Sometimes you need to format a string without displaying it in the console. For example, you might need to display formatted output in a graphical interface, such as a message dialog. When this is the case, you can use the `String.format` method.

The `String.format` method works exactly like the `System.out.printf` method, except that it does not display the formatted string on the screen. Instead, it returns a reference to the formatted string. You can assign the reference to a variable, and then use it later. Here is the method's general format:

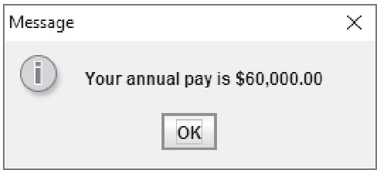
```
String.format(FormatString, ArgumentList)
```

In the general format, *`FormatString`* is a string that contains text, special formatting specifiers, or both. *`ArgumentList`* is a list of zero or more additional arguments, which will be formatted according to the format specifiers listed in the format string. The syntax for writing the format specifiers is the same as with the `System.out.printf` method. The method creates a string in memory that is formatted as specified, and returns a reference to that string. For example, look at the program in Code Listing 3-21. The program's output is shown in Figure 3-22.

**Code Listing 3-21** (CurrencyFormat2.java)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates how to use the String.format
   method to format a number as currency.
*/

public class CurrencyFormat2
{
   public static void main(String[] args)
   {
      double monthlyPay = 5000.0;
      double annualPay = monthlyPay * 12;
      String output = String.format("Your annual pay is $%,.2f", annualPay);

      JOptionPane.showMessageDialog(null, output);
   }
}
```

**Figure 3-22** Output of Code Listing 3-21 (Oracle Corporate Counsel)



Your annual pay is $60,000.00

Let's take a closer look at the program. Line 12 declares a `double` variable named `monthlyPay`, initialized with the value 5000.0, and line 13 declares a `double` variable named `annualPay`, initialized with the result of the calculation `monthlyPay * 12`. Line 14 declares a `String` variable named `output`, and initializes it with the string that is returned from the `String.format` method. In line 16, the `output` variable is passed as an argument to the `JOptionPane.showMessageDialog` method.

The program in Code Listing 3-21 can be simplified. We can combine the steps of calling the `String.format` method, and passing the value that it returns to the `JOptionPane.showMessageDialog` method. This allows us to eliminate the declaration of the `output` variable. Code Listing 3-22 shows how this is done. The program's output is the same as shown in Figure 3-22.

**Code Listing 3-22** (`CurrencyFormat3.java`)

```java
import javax.swing.JOptionPane;

/**
   This program demonstrates how to use the String.format
   method to format a number as currency.
*/

public class CurrencyFormat3
{
   public static void main(String[] args)
   {
      double monthlyPay = 5000.0;
      double annualPay = monthlyPay * 12;

      JOptionPane.showMessageDialog(null,
             String.format("Your annual pay is $%,.2f", annualPay));
   }
}
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.29 Assume the following variable declaration exists in a program:

```java
double number = 1234567.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable formatted as:

```
1,234,567.46
```

3.30 Assume the following variable declaration exists in a program:

```java
double number = 123.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable rounded to one decimal place, in a field that is 10 spaces wide. (Do not use comma separators.)

3.31 Assume the following variable declaration exists in a program:

```
double number = 123.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable padded with leading zeros, in a field that is eight spaces wide, rounded to one decimal place. (Do not use comma separators.)

3.32 Assume the following variable declaration exists in a program:

```
int number = 123456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable in a field that is 10 spaces wide, with comma separators.

3.33 Assume the following variable declaration exists in a program:

```
double number = 123456.789;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable left-justified, with comma separators, in a field that is 20 spaces wide, rounded to two decimal places.

3.34 Assume the following declaration exists in a program:

```
String name = "James";
```

Write a statement that uses `System.out.printf` to display the value of `name` in a field that is 20 spaces wide.

## 3.11 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics.

- **Using = instead of == to compare primitive values.** Remember, = is the assignment operator and == tests for equality.
- **Using == instead of the `equals` method to compare `String` objects.** You cannot use the == operator to compare the contents of a `String` object with another string. Instead you must use the `equals` or `compareTo` method.
- **Forgetting to enclose an `if` statement's `boolean` expression in parentheses.** Java requires that the `boolean` expression being tested by an `if` statement is enclosed in a set of parentheses. An error will result if you omit the parentheses or use any other grouping characters.
- **Writing a semicolon at the end of an `if` clause.** When you write a semicolon at the end of an `if` clause, Java assumes that the conditionally executed statement is a null or empty statement.
- **Forgetting to enclose multiple conditionally executed statements in braces.** Normally the `if` statement conditionally executes only one statement. To conditionally execute more than one statement, you must enclose them in braces.
- **Omitting the trailing `else` in an `if-else-if` statement.** This is not a syntax error, but can lead to logical errors. If you omit the trailing `else` from an `if-else-if`

statement, no code will be executed if none of the statement's `boolean` expressions are `true`.

- **Not writing complete `boolean` expressions on both sides of a logical `&&` or `||` operator.** You must write a complete `boolean` expression on both sides of a logical `&&` or `||` operator. For example, the expression `x > 0 && < 10` is not valid because `< 10` is not a complete expression. The expression should be written as `x > 0 && x < 10`.
- **Trying to perform case-insensitive string comparisons with the `String` class's `equals` and `compareTo` methods.** To perform case-insensitive string comparisons, use the `String` class's `equalsIgnoreCase` and `compareToIgnoreCase` methods.
- **Using a `SwitchExpression` that is not an `int`, `short`, `byte`, `char`, or `String`.** The `switch` statement can only evaluate expressions that are of the `int`, `short`, `byte`, `char`, or `String` data types.
- **Using a *`CaseExpression`* that is not a literal or a `final` variable.** Because the compiler must determine the value of a *`CaseExpression`* at compile time, *`CaseExpressions`* must be either literal values or `final` variables.
- **Forgetting to write a colon at the end of a `case` statement.** A colon must appear after the *`CaseExpression`* in each `case` statement.
- **Forgetting to write a `break` statement in a `case` section.** This is not a syntax error, but it can lead to logical errors. The program does not branch out of a `switch` statement until it reaches a `break` statement or the end of the `switch` statement.
- **Forgetting to write a `default` section in a `switch` statement.** This is not a syntax error, but can lead to a logical error. If you omit the `default` section, no code will be executed if none of the *`CaseExpressions`* match the *`SwitchExpression`*.
- **Reversing the `?` and the `:` when using the conditional operator.** When using the conditional operator, the `?` character appears first in the conditional expression, then the `:` character.
- **When formatting a number with `System.out.printf` or `String.format`, writing the flags, field width, and precision in an incorrect order.**
- **When writing a format specifier for the `System.out.printf` or `String.format` methods, using the wrong type indicator (`%f` = floating-point, `%d` = integer, `%s` = string).**

## Review Questions and Exercises

### Multiple Choice and True/False

1. The `if` statement is an example of a __________.
   a. sequence structure
   b. decision structure
   c. pathway structure
   d. class structure

2. This type of expression has a value of either `true` or `false`.
   a. `binary` expression
   b. `decision` expression
   c. `unconditional` expression
   d. `boolean` expression

3. `>`, `<`, and `==` are __________.
   a. relational operators
   b. logical operators
   c. conditional operators
   d. ternary operators
4. `&&`, `||`, and `!` are __________.
   a. relational operators
   b. logical operators
   c. conditional operators
   d. ternary operators
5. This is an empty statement that does nothing.
   a. missing statement
   b. virtual statement
   c. `null` statement
   d. conditional statement
6. To create a block of statements, you enclose the statements in these.
   a. parentheses `()`
   b. square brackets `[]`
   c. angled brackets `<>`
   d. braces `{}`
7. This is a `boolean` variable that signals when some condition exists in the program.
   a. flag
   b. signal
   c. sentinel
   d. siren
8. How does the character ‘A’ compare to the character ‘B’?
   a. ‘A’ is greater than ‘B’
   b. ‘A’ is less than ‘B’
   c. ‘A’ is equal to ‘B’
   d. You cannot compare characters
9. This is an `if` statement that appears inside another `if` statement.
   a. nested `if` statement
   b. tiered `if` statement
   c. dislodged `if` statement
   d. structured `if` statement
10. An `else` clause always goes with __________.
    a. the closest previous `if` clause that doesn’t already have its own `else` clause
    b. the closest `if` clause
    c. the `if` clause that is randomly selected by the compiler
    d. none of these
11. When determining whether a number is inside a range, it’s best to use this operator.
    a. `&&`
    b. `!`
    c. `||`
    d. `? :`

12. This determines whether two different `String` objects contain the same string.
    a. the `==` operator
    b. the `=` operator
    c. the `equals` method
    d. the `stringCompare` method
13. The conditional operator takes this many operands.
    a. one
    b. two
    c. three
    d. four
14. This section of a `switch` statement is branched to if none of the `case` expressions match the `switch` expression.
    a. `else`
    b. `default`
    c. `case`
    d. `otherwise`
15. You can use this method to display formatted output in a console window.
    a. `Format.out.println`
    b. `Console.format`
    c. `System.out.printf`
    d. `System.out.formatted`
16. **True or False:** The `=` operator and the `==` operator perform the same operation.
17. **True or False:** A conditionally executed statement should be indented one level from the `if` clause.
18. **True or False:** All lines in a conditionally executed block should be indented one level.
19. **True or False:** When an `if` statement is nested in the `if` clause of another statement, the only time the inner `if` statement is executed is when the `boolean` expression of the outer `if` statement is `true`.
20. **True or False:** When an `if` statement is nested in the `else` clause of another statement, the only time the inner `if` statement is executed is when the `boolean` expression of the outer `if` statement is `true`.
21. **True or False:** The scope of a variable is limited to the block in which it is defined.

### Find the Error

Find the errors in the following code:

1.
```
// Warning! This code contains ERRORS!
if (x == 1);
   y = 2;
else if (x == 2);
   y = 3;
else if (x == 3);
   y = 4;
```

2.
```
// Warning! This code contains an ERROR!
if (average = 100)
  System.out.println("Perfect Average!");
```

3.
```
// Warning! This code contains ERRORS!
if (num2 == 0)
  System.out.println("Division by zero is not possible.");
  System.out.println("Please run the program again ");
  System.out.println("and enter a number besides zero.");
else
  Quotient = num1 / num2;
  System.out.print("The quotient of " + Num1);
  System.out.print(" divided by " + Num2 + " is ");
  System.out.println(Quotient);
```

4.
```
// Warning! This code contains ERRORS!
switch (score)
{
   case (score > 90):
     grade = 'A';
     break;
   case(score > 80):
     grade = 'b';
     break;
   case(score > 70):
     grade = 'C';
     break;
   case (score > 60):
     grade = 'D';
     break;
   default:
     grade = 'F';
}
```

5. The following statement should determine whether `x` is not greater than 20. What is wrong with it?

```
if (!x > 20)
```

6. The following statement should determine whether `count` is within the range of 0 through 100. What is wrong with it?

```
if (count >= 0 || count <= 100)
```

7. The following statement should determine whether `count` is outside the range of 0 through 100. What is wrong with it?

```
if (count < 0 && count > 100)
```

8. The following statement should assign 0 to `z` if `a` is less than 10; otherwise, it should assign 7 to `z`. What is wrong with it?

```
z = (a < 10) : 0 ? 7;
```

9. Assume that `partNumber` references a `String` object. The following `if` statement should perform a case-insensitive comparison. What is wrong with it?

```
if (partNumber.equals("BQ789W4"))
   available = true;
```

10. What is wrong with the following code?

```
double value = 12345.678;
System.out.printf("%.2d", value);
```

## Algorithm Workbench

1. Write an `if` statement that assigns 100 to `x` when `y` is equal to 0.
2. Write an `if-else` statement that assigns 0 to `x` when `y` is equal to 10. Otherwise, it should assign 1 to `x`.
3. Using the following chart, write an `if-else-if` statement that assigns .10, .15, or .20 to `commission`, depending on the value in `sales`.

| Sales | Commission Rate |
|---|---|
| Up to $10,000 | 10% |
| $10,000 to $15,000 | 15% |
| Over $15,000 | 20% |

4. Write an `if` statement that sets the variable `hours` to 10 when the `boolean` flag variable `minimum` is equal to `true`.
5. Write nested `if` statements that perform the following tests: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of the two.
6. Write an `if` statement that prints the message “The number is valid” if the variable `grade` is within the range 0 through 100.
7. Write an `if` statement that prints the message “The number is valid” if the variable `temperature` is within the range −50 through 150.
8. Write an `if` statement that prints the message “The number is not valid” if the variable `hours` is outside the range 0 through 80.
9. Write an `if-else` statement that displays the `String` objects `title1` and `title2` in alphabetical order.
10. Convert the following `if-else-if` statement into a `switch` statement:

```
if (choice == 1)
{
  System.out.println("You selected 1.");
}
else if (choice == 2 || choice == 3)
```

```
{
  System.out.println("You selected 2 or 3.");
}
else if (choice == 4)
{
   System.out.println("You selected 4.");
}
else
{
   System.out.println("Select again please.");
}
```

11. Match the conditional expression with the `if-else` statement that performs the same operation.

    a. `q = x < y ? a + b : x * 2;`
    b. `q = x < y ? x * 2 : a + b;`
    c. `q = x < y ? 0 : 1;`

```
____      if (x < y)
              q = 0;
          else
              q = 1;
____      if (x < y)
              q = a + b;
          else
              q = x * 2;
____      if (x < y)
              q = x * 2;
          else
              q = a + b;
```

12. Assume the `double` variable `number` contains the value 12345.6789. Write a statement that uses `System.out.printf` to display the number as 12345.7.
13. Assume the `double` variable `number` contains the value 12345.6789. Write a statement that uses `System.out.printf` to display the number as 12,345.68.
14. Assume the `int` variable `number` contains the value 1234567. Write a statement that uses `System.out.printf` to display the number as 1,234,567.

### Short Answer

1. Explain what is meant by the phrase “conditionally executed.”
2. Explain why a misplaced semicolon can cause an `if` statement to operate incorrectly.
3. Why is it good advice to indent all the statements inside a set of braces?
4. What happens when you compare two `String` objects with the `==` operator?
5. Explain the purpose of a flag variable. Of what data type should a flag variable be?
6. What risk does a programmer take when not placing a trailing `else` at the end of an `if-else-if` statement?

7. Briefly describe how the && operator works.
8. Briefly describe how the || operator works.
9. Why are the relational operators called “relational”?
10. When does a constructor execute? What is its purpose?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Roman Numerals

Write a program that prompts the user to enter a number within the range of 1 through 10. The program should display the Roman numeral version of that number. If the number is outside the range of 1 through 10, the program should display an error message.

### 2. Magic Dates

The date June 10, 1960, is special because when we write it in the following format, the month times the day equals the year:

6/10/60

Write a program that asks the user to enter a month (in numeric form), a day, and a two-digit year. The program should then determine whether the month times the day is equal to the year. If so, it should display a message saying the date is magic. Otherwise, it should display a message saying the date is not magic.

### 3. Body Mass Index

Write a program that calculates and displays a person’s body mass index (BMI). The BMI is often used to determine whether a person is overweight or underweight for his or her height. A person’s BMI is calculated with the following formula:

$$BMI = Weight \times 703/Height^2$$

where *weight* is measured in pounds and *height* is measured in inches. The program should display a message indicating whether the person has optimal weight, is underweight, or is overweight. A sedentary person’s weight is considered optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered underweight. If the BMI value is greater than 25, the person is considered overweight.

### 4. Test Scores and Grade

Write a program that has variables to hold three test scores. The program should ask the user to enter three test scores and then assign the values entered to the variables. The program should display the average of the test scores and the letter grade that is assigned for the test score average. Use the grading scheme in the following table:

| Test Score Average | Letter Grade |
|---|---|
| 90–100 | A |
| 80–89 | B |
| 70–79 | C |
| 60–69 | D |
| Below 60 | F |

### 5. Mass and Weight

Scientists measure an object's mass in kilograms and its weight in Newtons. If you know the amount of mass that an object has, you can calculate its weight, in Newtons, with the following formula:

$$Weight = Mass \times 9.8$$

Write a program that asks the user to enter an object's mass, and then calculate its weight. If the object weighs more than 1,000 Newtons, display a message indicating that it is too heavy. If the object weighs less than 10 Newtons, display a message indicating that the object is too light.

### 6. Time Calculator

VideoNote
The Time Calculator Problem

Write a program that asks the user to enter a number of seconds.

- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days in that many seconds.

### 7. Sorted Names

Write a program that asks the user to enter three names, and then displays the names sorted in ascending order. For example, if the user entered "Charlie", "Leslie", and "Andy", the program would display:

```
Andy
Charlie
Leslie
```

### 8. Software Sales

A software company sells a package that retails for $99. Quantity discounts are given according to the following table:

| Quantity | Discount |
|---|---|
| 10–19 | 20% |
| 20–49 | 30% |

| Quantity | Discount |
|---|---|
| 50–99 | 40% |
| 100 or more | 50% |

Write a program that asks the user to enter the number of packages purchased. The program should then display the amount of the discount (if any) and the total amount of the purchase after the discount.

### 9. Shipping Charges

The Fast Freight Shipping Company charges the following rates:

| Weight of Package | Rate per 500 Miles Shipped |
|---|---|
| 2 pounds or less | \$1.10 |
| Over 2 pounds but not more than 6 pounds | \$2.20 |
| Over 6 pounds but not more than 10 pounds | \$3.70 |
| Over 10 pounds | \$3.80 |

The shipping charges per 500 miles are not prorated. For example, if a 2-pound package is shipped 550 miles, the charges would be \$2.20. Write a program that asks the user to enter the weight of a package and then displays the shipping charges.

### 10. Fat Gram Calculator

Write a program that asks the user to enter the number of calories and fat grams in a food item. The program should display the percentage of the calories that come from fat. One gram of fat has 9 calories; therefore:

*Calories from fat* = *Fat grams* * 9

The percentage of calories from fat can be calculated as follows:

*Calories from fat* ÷ *Total calories*

If the calories from fat are less than 30 percent of the total calories of the food, it should also display a message indicating the food is low in fat.

**NOTE:** The number of calories from fat cannot be greater than the total number of calories in the food item. If the program determines that the number of calories from fat is greater than the number of calories in the food item, it should display an error message indicating that the input is invalid.

### 11. Running the Race

Write a program that asks for the names of three runners and the time, in minutes, it took each of them to finish a race. The program should display the names of the runners in the order that they finished.

## 12. The Speed of Sound

The following table shows the approximate speed of sound in air, water, and steel:

| Medium | Speed |
|---|---|
| Air | 1,100 feet per second |
| Water | 4,900 feet per second |
| Steel | 16,400 feet per second |

Write a program that asks the user to enter "air", "water", or "steel", and the distance that a sound wave will travel in the medium. The program should then display the amount of time it will take. You can calculate the amount of time it takes sound to travel in air with the following formula:

$$Time = Distance/1{,}100$$

You can calculate the amount of time it takes sound to travel in water with the following formula:

$$Time = Distance/4{,}900$$

You can calculate the amount of time it takes sound to travel in steel with the following formula:

$$Time = Distance/16{,}400$$

## 13. Mobile Service Provider

A mobile phone service provider has three different subscription packages for its customers:

| | |
|---|---|
| Package A: | For $39.99 per month, 450 minutes are provided. Additional minutes are $0.45 per minute. |
| Package B: | For $59.99 per month, 900 minutes are provided. Additional minutes are $0.40 per minute. |
| Package C: | For $69.99 per month, unlimited minutes are provided. |

Write a program that calculates a customer's monthly bill. It should ask the user to enter the letter of the package the customer has purchased (A, B, or C) and the number of minutes that were used. The program should display the total charges.

## 14. Mobile Service Provider, Part 2

Modify the program you wrote for Programming Challenge 13 so it also calculates and displays the amount of money Package A customers would save if they purchased Package B or C, and the amount of money Package B customers would save if they purchased Package C. If there would be no savings, no message should be printed.

### 15. Bank Charges

A bank charges a base fee of $10 per month, plus the following check fees for a commercial checking account:

$.10 each for less than 20 checks
$.08 each for 20–39 checks
$.06 each for 40–59 checks
$.04 each for 60 or more checks

Write a program that asks for the number of checks written for the month. The program should then calculate and display the bank's service fees for the month.

### 16. Book Club Points

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 1 book, he or she earns 5 points.
- If a customer purchases 2 books, he or she earns 15 points.
- If a customer purchases 3 books, he or she earns 30 points.
- If a customer purchases 4 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books that he or she has purchased this month and then displays the number of points awarded.

### 17. Wi-Fi Diagnostic Tree

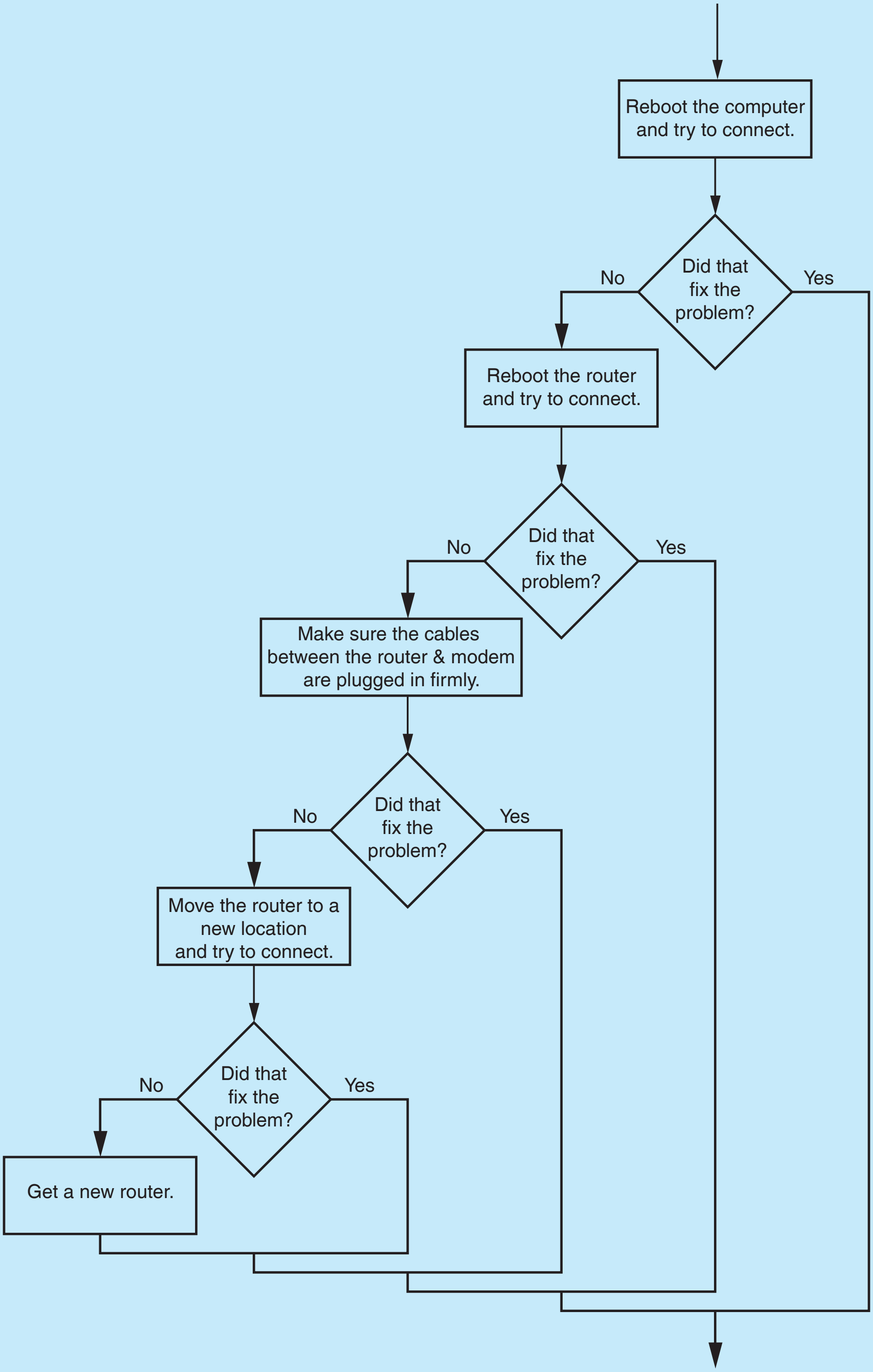
Figure 3-23 shows a simplified flowchart for troubleshooting a bad Wi-Fi connection. Use the flowchart to create a program that leads a person through the steps of fixing a bad Wi-Fi connection. Here is an example of the program's output:

```
Reboot the computer and try to connect.
Did that fix the problem?  no [Enter]
Reboot the router and try to connect.
Did that fix the problem?  yes [Enter]
```

Notice that the program ends as soon as a solution is found to the problem. Here is another example of the program's output:

```
Reboot the computer and try to connect.
Did that fix the problem?  no [Enter]
Reboot the router and try to connect.
Did that fix the problem?  no [Enter]
Make sure the cables between the router & modem are plugged in firmly.
Did that fix the problem?  no [Enter]
Move the router to a new location.
Did that fix the problem?  no [Enter]
Get a new router.
```

**Figure 3-23** Troubleshooting a bad Wi-Fi connection


Reboot the computer and try to connect.
Did that fix the problem?
No
Yes
Reboot the router and try to connect.
Did that fix the problem?
No
Yes
Make sure the cables between the router & modem are plugged in firmly.
Did that fix the problem?
No
Yes
Move the router to a new location and try to connect.
Did that fix the problem?
No
Yes
Get a new router.

### 18. Restaurant Selector

You have a group of friends coming to visit for your high school reunion, and you want to take them out to eat at a local restaurant. You aren't sure if any of them have dietary restrictions, but your restaurant choices are as follows:

*Joe's Gourmet Burgers* – Vegetarian: No, Vegan: No, Gluten-Free: No
*Main Street Pizza Company* – Vegetarian: Yes, Vegan: No, Gluten-Free: Yes
*Corner Café* – Vegetarian: Yes, Vegan: Yes, Gluten-Free: Yes
*Mama's Fine Italian* – Vegetarian: Yes, Vegan: No, Gluten-Free: No
*The Chef's Kitchen* – Vegetarian: Yes, Vegan: Yes, Gluten-Free: Yes

Write a program that asks whether any members of your party are vegetarian, vegan, or gluten-free, and then displays only the restaurants that you may take the group to. Here is an example of the program's output:

```
Is anyone in your party a vegetarian? yes [Enter]
Is anyone in your party a vegan? no [Enter]
Is anyone in your party gluten-free? yes [Enter]
Here are your restaurant choices:
   Main Street Pizza Company
   Corner Cafe
   The Chef's Kitchen
```

Here is another example of the program's output:

```
Is anyone in your party a vegetarian? yes [Enter]
Is anyone in your party a vegan? yes [Enter]
Is anyone in your party gluten-free? yes [Enter]
Here are your restaurant choices:
   Corner Cafe
   The Chef's Kitchen
```

CHAPTER

# 4 Loops and Files

**TOPICS**

4.1 The Increment and Decrement Operators
4.2 The `while` Loop
4.3 Using the `while` Loop for Input Validation
4.4 The `do-while` Loop
4.5 The `for` Loop
4.6 Running Totals and Sentinel Values
4.7 Nested Loops
4.8 The `break` and `continue` Statements (Optional)
4.9 Deciding Which Loop to Use
4.10 Introduction to File Input and Output
4.11 Generating Random Numbers with the `Random` Class
4.12 Common Errors to Avoid

## 4.1 The Increment and Decrement Operators

**CONCEPT: ++ and -- are operators that add and subtract one from their operands.**

To *increment* a value means to increase it by one, and to *decrement* a value means to decrease it by one. Both of the following statements increment the variable `number`:

```
number = number + 1;
number += 1;
```

And `number` is decremented in both of the following statements:

```
number = number − 1;
number −= 1;
```

Java provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is `++` and the decrement operator is `−−`. The following statement uses the `++` operator to increment `number`:

```
number++;
```

And the following statement decrements `number`:

```
number−−;
```

**NOTE:** The expression `number++` is pronounced “number plus plus,” and `number--` is pronounced “number minus minus.”

The program in Code Listing 4-1 demonstrates the `++` and `--` operators.

**Code Listing 4-1** `(IncrementDecrement.java)`

```java
/**
   This program demonstrates the ++ and -- operators.
*/

public class IncrementDecrement
{
   public static void main(String[] args)
   {
      int number = 4; // number starts out with 4

      // Display the value in number.
      System.out.println("number is " + number);
      System.out.println("I will increment number.");

      // Increment number.
      number++;

      // Display the value in number again.
      System.out.println("Now, number is " + number);
      System.out.println("I will decrement number.");

      // Decrement number.
      number--;

      // Display the value in number once more.
      System.out.println("Now, number is " + number);
   }
}
```

**Program Output**

```
number is 4
I will increment number.
Now, number is 5
I will decrement number.
Now, number is 4
```

The statements in Code Listing 4-1 show the increment and decrement operators used in *postfix mode*, which means the operator is placed after the variable. The operators also work in *prefix mode*, where the operator is placed before the variable name as follows:

```
++number;

--number;
```

In both postfix and prefix mode, these operators add one to or subtract one from their operand. Code Listing 4-2 demonstrates this.

**Code Listing 4-2** `(Prefix.java)`

```
/**
   This program demonstrates the ++ and -- operators
   in prefix mode.
*/

public class Prefix
{
   public static void main(String[] args)
   {
      int number = 4; // number starts out with 4

      // Display the value in number.
      System.out.println("number is " + number);
      System.out.println("I will increment number.");

      // Increment number.
      ++number;

      // Display the value in number again.
      System.out.println("Now, number is " + number);
      System.out.println("I will decrement number.");

      // Decrement number.
      --number;

      // Display the value in number once again.
      System.out.println("Now, number is " + number);
   }
}
```

**Program Output**

```
number is 4
I will increment number.
Now, number is 5
I will decrement number.
Now, number is 4
```

## The Difference between Postfix and Prefix Modes

In Code Listings 4-1 and 4-2, the statements `number++` and `++number` increment the variable `number`, while the statements `number--` and `--number` decrement the variable `number`. In these simple statements, it doesn't matter whether the operator is used in postfix or prefix mode. The difference is important, however, when these operators are used in statements that do more than just increment or decrement. For example, look at the following code:

```
number = 4;
System.out.println(number++);
```

The statement that calls the `println` method does two things: (1) calls `println` to display the value of `number`, and (2) increments `number`. But which happens first? The `println` method will display a different value if `number` is incremented first than if `number` is incremented last. The answer depends upon the mode of the increment operator.

Postfix mode causes the increment to happen after the value of the variable is used in the expression. In the previously shown statement, the `println` method will display 4 and then `number` will be incremented to 5. Prefix mode, however, causes the increment to happen first. Here is an example:

```
number = 4;
System.out.println(++number);
```

In these statements, `number` is incremented to 5, then `println` will display the value in `number` (which is 5). For another example, look at the following code:

```
int x = 1, y;
y = x++;              // Postfix increment
```

The first statement declares the variable `x` (initialized with the value 1) and the variable `y`. The second statement does the following:

- It assigns the value of `x` to the variable `y`.
- The variable `x` is incremented.

The value that will be stored in `y` depends on when the increment takes place. Because the `++` operator is used in postfix mode, it acts after the assignment takes place. So, this code will store 1 in `y`. After the code has executed, `x` will contain 2. Let's look at the same code, but with the `++` operator used in prefix mode as follows:

```
int x = 1, y;
y = ++x;              // Prefix increment
```

The first statement declares the variable `x` (initialized with the value 1) and the variable `y`. In the second statement, the `++` operator is used in prefix mode, so it acts on the variable before the assignment takes place. So, this code will store 2 in `y`. After the code has executed, `x` will also contain 2.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

4.1 What will the following program segments display?

a)
```
 x = 2;
 y = x++;
System.out.println(y);
```

b)
```
 x = 2;
System.out.println(x++);
```

c)
```
 x = 2;
System.out.println(--x);
```

d)
```
 x = 8;
y = x--;
System.out.println(y);
```

## 4.2 The `while` Loop

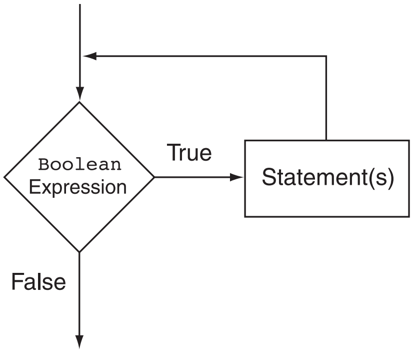
**CONCEPT:** A loop is part of a program that repeats.

In Chapter 3, you were introduced to the concept of control structures, which direct the flow of a program. A *loop* is a control structure that causes a statement or group of statements to repeat. Java has three looping control structures: the `while` loop, the `do-while` loop, and the `for` loop. The difference among each of these is how they control the repetition. In this section we will focus on the `while` loop.

**VideoNote** The while Loop

The `while` loop has two important parts: (1) a `boolean` expression that is tested for a `true` or `false` value, and (2) a statement or block of statements that is repeated as long as the expression is `true`. Figure 4-1 shows the logic of a `while` loop.

**Figure 4-1** Logic of a `while` loop



Here is the general format of the `while` loop:

```
while (BooleanExpression)
    Statement;
```

In the general format, *`BooleanExpression`* is any valid `boolean` expression, and *`Statement`* is any valid Java statement. The first line shown in the format is sometimes called the *loop header*. It consists of the key word `while` followed by the *`BooleanExpression`* enclosed in parentheses.

Here's how the loop works: The *`BooleanExpression`* is tested, and if it is `true`, the *`Statement`* is executed. Then, the *`BooleanExpression`* is tested again. If it is `true`, the *`Statement`* is executed. This cycle repeats until the *`BooleanExpression`* is `false`.

The statement that is repeated is known as the *body* of the loop. It is also considered a conditionally executed statement, because it is only executed under the condition that the *`BooleanExpression`* is `true`.

Notice there is no semicolon at the end of the loop header. Like the `if` statement, the `while` loop is not complete without the conditionally executed statement that follows it.

If you wish the `while` loop to repeat a block of statements, the format is as follows:

```
while (BooleanExpression)
{
   Statement;
   Statement;
   // Place as many statements here
   // as necessary.
}
```

The `while` loop works like an `if` statement that executes over and over. As long as the expression in the parentheses is `true`, the conditionally executed statement or block will repeat. The program in Code Listing 4-3 uses the `while` loop to print "Hello" five times.

**Code Listing 4-3** **(`WhileLoop.java`)**

```
/**
   This program demonstrates the while loop.
*/

public class WhileLoop
{
   public static void main(String[] args)
   {
      int number = 1;

      while (number <= 5)
      {
         System.out.println("Hello");
         number++;
      }
```

```

        System.out.println("That's all!");
    }
}
```

**Program Output**

```
Hello
Hello
Hello
Hello
Hello
That's all!
```

Let's take a closer look at this program. An integer variable, `number`, is declared and initialized with the value 1. The `while` loop begins with the following statement:

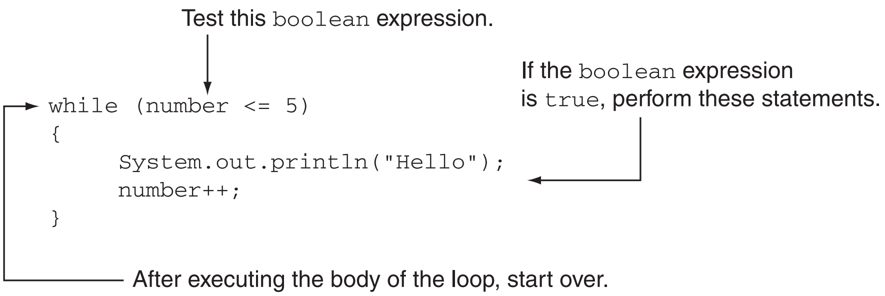
```
while (number <= 5)
```

This statement tests the variable `number` to determine whether it is less than or equal to 5. If it is, then the statements in the body of the loop are executed as follows:

```
System.out.println("Hello");
number++;
```

The first statement in the body of the loop prints the word "Hello". The second statement uses the increment operator to add one to `number`. This is the last statement in the body of the loop, so after it executes, the loop starts over. It tests the `boolean` expression again, and if it is `true`, the statements in the body of the loop are executed. This cycle repeats until the `boolean` expression `number <= 5` is `false`, as illustrated in Figure 4-2.

**Figure 4-2** The `while` loop



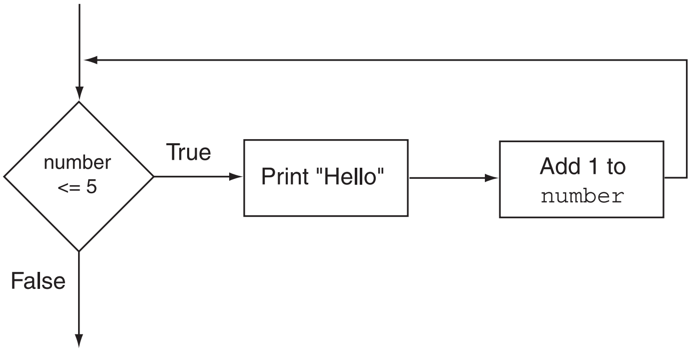
```
while (number <= 5)
{
     System.out.println("Hello");
     number++;
}
```

Test this boolean expression.
If the boolean expression is true, perform these statements.
After executing the body of the loop, start over.

Each repetition of a loop is known as an *iteration*. This loop will perform five iterations because the variable `number` is initialized with the value 1, and it is incremented each time the body of the loop is executed. When the expression `number <= 5` is tested and found to be `false`, the loop will terminate and the program will resume execution at the statement that immediately follows the loop. Figure 4-3 shows the logic of this loop.

In this example, the `number` variable is referred to as the *loop control variable* because it controls the number of times that the loop iterates.

**Figure 4-3** Logic of the example `while` loop



## The `while` Loop Is a Pretest Loop

The `while` loop is known as a *pretest* loop, which means it tests its expression before each iteration. Notice the variable declaration of `number` in Code Listing 4-3:

```
int number = 1;
```

The `number` variable is initialized with the value 1. If `number` had been initialized with a value that is greater than 5, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number <= 5)
{
   System.out.println("Hello");
   number++;
}
```

An important characteristic of the `while` loop is that the loop will never iterate if the `boolean` expression is `false` to start with. If you want to be sure that a `while` loop executes the first time, you must initialize the relevant data in such a way that the `boolean` expression starts out as `true`.

## Infinite Loops

In all but rare cases, loops must contain a way to terminate within themselves. This means that something inside the loop must eventually make the `boolean` expression `false`. The loop in Code Listing 4-3 stops when the expression `number <= 5` is `false`.

If a loop does not have a way of stopping, it is called an *infinite loop* as it continues to repeat until the program is interrupted. Here is an example of an infinite loop:

```
int number = 1;
while (number <= 5)
{
   System.out.println("Hello");
}
```

This is an infinite loop because it does not contain a statement that changes the value of the `number` variable. Each time the `boolean` expression is tested, `number` will contain the value 1.

It's also possible to create an infinite loop by accidentally placing a semicolon after the first line of the `while` loop. Here is an example:

```
int number = 1;
while (number <= 5); // This semicolon is an ERROR!
{
   System.out.println("Hello");
   number++;
}
```

The semicolon at the end of the first line is assumed to be a `null` statement and disconnects the `while` statement from the block that comes after it. To the compiler, this loop looks like the following:

```
while (number <= 5);
```

This `while` loop will forever execute the null statement, which does nothing. The program will appear to have "gone into space" because there is nothing to display screen output or show activity.

## Don't Forget the Braces with a Block of Statements

If you are using a block of statements, don't forget to enclose all of the statements in a set of braces. If the braces are accidentally left out, the `while` statement conditionally executes only the very next statement. For example, look at the following code:

```
int number = 1;
// This loop is missing its braces!
while (number <= 5)
   System.out.println("Hello");
   number++;
```

In this code the `number++` statement is not in the body of the loop. Because the braces are missing, the `while` statement executes only the statement that immediately follows it. This loop will execute infinitely because there is no code in its body that changes the `number` variable.

## Programming Style and the `while` Loop

It’s possible to create loops that look like the following:

```
while (number <= 5) { System.out.println("Hello"); number++; }
```

Avoid this style of programming. The programming style you should use with the `while` loop is similar to that of the `if` statement as follows:

- If there is only one statement repeated by the loop, it should appear on the line after the `while` statement and be indented one additional level. The statement can optionally appear inside a set of braces.
- If the loop repeats a block, each line inside the braces should be indented.

This programming style should visually set the body of the loop apart from the surrounding code. In general, you’ll find a similar style being used with the other types of loops presented in this chapter.

## In the Spotlight:

### Designing a Program with a `while` Loop

A project currently underway at Chemical Labs, Inc., requires that a substance be continually heated in a vat. A technician must check the substance’s temperature every 15 minutes. If the substance’s temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat’s thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. *Prompt the user to enter the substance’s temperature.*
2. *Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:*
    (a) *Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.*
    (b) *Prompt the user to enter the substance’s temperature.*
3. *After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.*

After reviewing this algorithm, you realize that Steps 2(a) and 2(b) should not be performed if the test condition (temperature is greater than 102.5) is `false` to begin with. The `while` loop will work well in this situation, because it will not execute even once if its condition is `false`. Code Listing 4-4 shows the program.

**Code Listing 4-4** (`CheckTemperature.java`)

```
import java.util.Scanner;

```

```java
/**
   This program assists a technician in the process
   of checking a substance's temperature.
*/
public class CheckTemperature
{
   public static void main(String[] args)
   {
      final double MAX_TEMP = 102.5;   // Maximum temperature
      double temperature;              // To hold the temperature

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the current temperature.
      System.out.print("Enter the substance's Celsius temperature: ");
      temperature = keyboard.nextDouble();

      // As long as necessary, instruct the technician
      // to adjust the temperature.
      while (temperature > MAX_TEMP)
      {
         System.out.println("The temperature is too high. Turn the");
         System.out.println("thermostat down and wait 5 minutes.");
         System.out.println("Then, take the Celsius temperature again");
         System.out.print("and enter it here: ");
         temperature = keyboard.nextDouble();
      }

      // Remind the technician to check the temperature
      // again in 15 minutes.
      System.out.println("The temperature is acceptable.");
      System.out.println("Check it again in 15 minutes.");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the substance's Celsius temperature: 104.7 [Enter]
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then, take the Celsius temperature again
and enter it here: 103.2 [Enter]
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then, take the Celsius temperature again
and enter it here: 102.1 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

4.2 How many times will `"Hello World"` be printed in the following program segment?

```
int count = 10;
while (count < 1)
{
    System.out.println("Hello World");
    count++;
}
```

4.3 How many times will `"I love Java programming!"` be printed in the following program segment?

```
int count = 0;
while (count < 10)
   System.out.println("I love Java programming!);
```

## 4.3 Using the `while` Loop for Input Validation

**CONCEPT: The `while` loop can be used to create input routines that repeat until acceptable data is entered.**

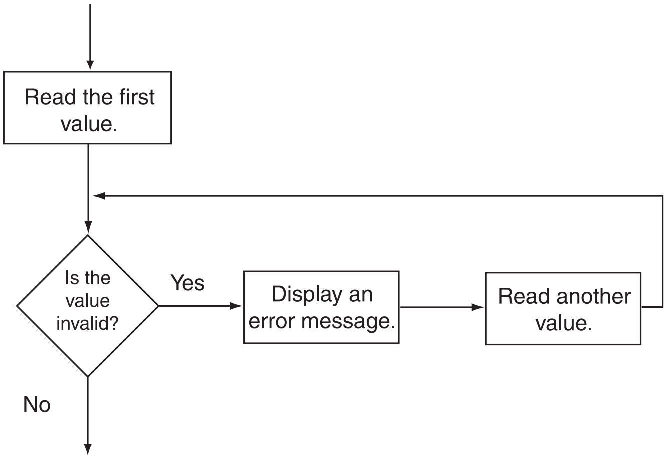
Perhaps the most famous saying of the computer industry is “garbage in, garbage out.” The integrity of a program’s output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting data given to a program by the user and determining whether it is valid. A good program should give clear instructions about the kind of input that is acceptable, and not assume the user has followed those instructions.

The `while` loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user reenter it as many times as necessary. For example, the loop in the following code asks for a number in the range of 1 through 100:

```
int number;

// Create a Scanner object for keyboard input.
Scanner keyboard = new Scanner(System.in);

// Get a number from the user.
System.out.print("Enter a number in the range of 1 through 100: ");
number = keyboard.nextInt();

// Validate the input.
while (number < 1 || number > 100)
{
   System.out.print("Invalid input. Enter a number in the range " +
                    "of 1 through 100: ");
   number = keyboard.nextInt();
}
```

This code first allows the user to enter a number. This takes place just before the loop. If the input is valid, the loop will not execute. If the input is invalid, however, the loop will display an error message and require the user to enter another number. The loop will continue to execute until the user enters a valid number. The general logic of performing input validation is shown in Figure 4-4.

**Figure 4-4** Input validation logic



The read operation that takes place just before the loop is called a *priming read*. It provides the first value for the loop to test. Subsequent values are obtained by the loop.

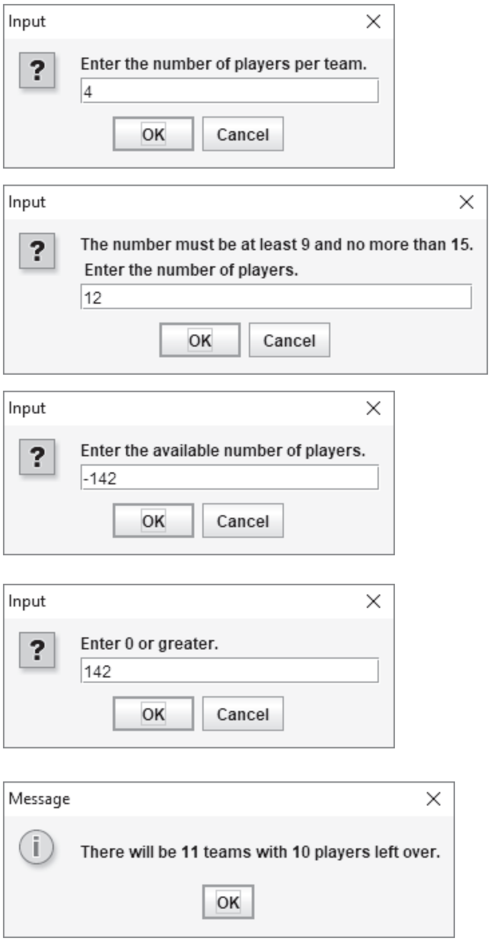
The program in Code Listing 4-5 calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses `while` loops (in lines 28 through 36 and lines 44 through 49) to validate the user's input. Figure 4-5 shows an example of interaction with the program.

**Code Listing 4-5** (`SoccerTeams.java`)

```
 1  import javax.swing.JOptionPane;
 2
 3  /**
 4     This program calculates the number of soccer teams
 5     that a youth league may create from the number of
 6     available players. Input validation is demonstrated
 7     with while loops.
 8  */
 9
10  public class SoccerTeams
11  {
12     public static void main(String[] args)
13     {
14        final int MIN_PLAYERS = 9;      // Minimum players per team
15        final int MAX_PLAYERS = 15;     // Maximum players per team
16        int players;                    // Number of available players
```

```
      int teamSize;                  // Number of players per team
      int teams;                     // Number of teams
      int leftOver;                  // Number of leftover players
      String input;                  // To hold the user input

      // Get the number of players per team.
      input = JOptionPane.showInputDialog("Enter the number of " +
                                             "players per team.");
      teamSize = Integer.parseInt(input);

      // Validate the number entered.
      while (teamSize < MIN_PLAYERS || teamSize > MAX_PLAYERS)
      {
         input = JOptionPane.showInputDialog("The number must " +
                                      "be at least " + MIN_PLAYERS +
                                      " and no more than " +
                                      MAX_PLAYERS + ".\n Enter " +
                                      "the number of players.");
         teamSize = Integer.parseInt(input);
      }

      // Get the number of available players.
      input = JOptionPane.showInputDialog("Enter the available " +
                                              "number of players.");
      players = Integer.parseInt(input);

      // Validate the number entered.
      while (players < 0)
      {
        input = JOptionPane.showInputDialog("Enter 0 or " +
                                                "greater.");
        players = Integer.parseInt(input);
     }

      // Calculate the number of teams.
      teams = players / teamSize;

      // Calculate the number of leftover players.
      leftOver = players % teamSize;

      // Display the results.
      JOptionPane.showMessageDialog(null, "There will be " +
                                        teams + " teams with " +
                                        leftOver +
                                        " players left over.");
     System.exit(0);
   }
}
```

**Figure 4-5** Interaction with the `SoccerTeams` program (Oracle Corporate Counsel)



This input dialog box appears first. The user enters 4 (an invalid value) and clicks the OK button.

Enter the number of players per team.

4

This input dialog box appears next. The user enters 12 and clicks the OK button.

The number must be at least 9 and no more than 15.
Enter the number of players.

12

This input dialog box appears next. The user enters –142 (an invalid value) and clicks the OK button.

Enter the available number of players.

-142

This input dialog box appears next. The user enters 142 and clicks the OK button.

Enter 0 or greater.

142

This message dialog box appears next.

There will be 11 teams with 10 players left over.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.4 Write an input validation loop that asks the user to enter a number in the range of 10 through 24.

4.5 Write an input validation loop that asks the user to enter ‘Y’, ‘y’, ‘N’, or ‘n’.

4.6 Write an input validation loop that asks the user to enter “Yes” or “No”.

## 4.4 The `do-while` Loop

**CONCEPT: The `do-while` loop is a posttest loop, which means its `boolean` expression is tested after each iteration.**

The `do-while` loop looks something like an inverted `while` loop. Here is the `do-while` loop's format when the body of the loop contains only a single statement:

```
do
    Statement;
while (BooleanExpression);
```

Here is the format of the `do-while` loop when the body of the loop contains multiple statements:

```
do
{
   Statement;
   Statement;
   // Place as many statements here as necessary.
} while (BooleanExpression);
```

**NOTE:** The `do-while` loop must be terminated with a semicolon.

The `do-while` loop is a *posttest* loop. This means it does not test its `boolean` expression until it has completed an iteration. As a result, the `do-while` loop always performs at least one iteration, even if the `boolean` expression is `false` to begin with. This differs from the behavior of a `while` loop, which you will recall is a pretest loop. For example, in the following `while` loop the `println` statement will not execute at all:

```
int x = 1;
while (x < 0)
   System.out.println(x);
```

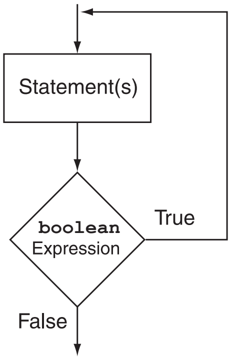
But the `println` statement in the following `do-while` loop will execute once because the `do-while` loop does not evaluate the expression `x < 0` until the end of the iteration:

```
int x = 1;
do
   System.out.println(x);
while (x < 0);
```

Figure 4-6 illustrates the logic of the `do-while` loop.

You should use the `do-while` loop when you want to make sure the loop executes at least once. For example, the program in Code Listing 4-6 averages a series of three test scores for a student. After the average is displayed, it asks the user whether he or she wants to average another set of test scores. The program repeats as long as the user enters `Y` for yes.

**Figure 4-6** Logic of the `do-while` loop



**Code Listing 4-6** (TestAverage1.java)

```
import java.util.Scanner;

/**
   This program demonstrates a user controlled loop.
*/

public class TestAverage1
{
   public static void main(String[] args)
   {
      int score1, score2, score3;      // Three test scores
      double average;                  // Average test score
      char repeat;                     // To hold 'y' or 'n'
      String input;                    // To hold input

      System.out.println("This program calculates the " +
                         "average of three test scores.");

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get as many sets of test scores as the user wants.
      do
      {
         // Get the first test score in this set.
         System.out.print("Enter score #1: ");
         score1 = keyboard.nextInt();

         // Get the second test score in this set.
         System.out.print("Enter score #2: ");
         score2 = keyboard.nextInt();

```

```
        // Get the third test score in this set.
        System.out.print("Enter score #3: ");
        score3 = keyboard.nextInt();

        // Consume the remaining newline.
        keyboard.nextLine();

        // Calculate and print the average test score.
        average = (score1 + score2 + score3) / 3.0;
        System.out.println("The average is " + average);
        System.out.println(); // Prints a blank line

        // Does the user want to average another set?
        System.out.println("Would you like to average " +
                           "another set of test scores?");
        System.out.print("Enter Y for yes or N for no: ");
        input = keyboard.nextLine();    // Read a line.
        repeat = input.charAt(0);       // Get the first char.

    } while (repeat == 'Y' || repeat == 'y');
  }
}
```

**Program Output with Example Input Shown in Bold**

```
This program calculates the average of three test scores.
Enter score #1: 89 [Enter]
Enter score #2: 90 [Enter]
Enter score #3: 97 [Enter]
The average is 92.0

Would you like to average another set of test scores?
Enter Y for yes or N for no: y [Enter]
Enter score #1: 78 [Enter]
Enter score #2: 65 [Enter]
Enter score #3: 88 [Enter]
The average is 77.0

Would you like to average another set of test scores?
Enter Y for yes or N for no: n [Enter]
```

When this program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user whether he or she wants to repeat the process. This type of loop is known as a *user controlled loop,* because it allows the user to decide the number of iterations.

## 4.5 The `for` Loop

**CONCEPT: The `for` loop is ideal for performing a known number of iterations.**

In general, there are two categories of loops: conditional loops and count-controlled loops. A *conditional loop* executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.

Sometimes you do know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a *count-controlled loop*. For example, if a loop asks the user to enter the sales amounts for each month in the year, it will iterate 12 times. In essence, the loop counts to 12 and asks the user to enter a sales amount each time it makes a count.

A count-controlled loop must possess three elements:

1. It must initialize a control variable to a starting value.
2. It must test the control variable by comparing it to a maximum value. When the control variable reaches its maximum value, the loop terminates.
3. It must update the control variable during each iteration. This is usually done by incrementing the variable.

In Java, the `for` loop is ideal for writing count-controlled loops. It is specifically designed to initialize, test, and update a loop control variable. Here is the format of the `for` loop when used to repeat a single statement:

```
for (Initialization; Test; Update)
    Statement;
```

The format of the `for` loop when used to repeat a block is as follows:

```
for (Initialization; Test; Update)
{
   Statement;
   Statement;
   // Place as many statements here as necessary.
}
```

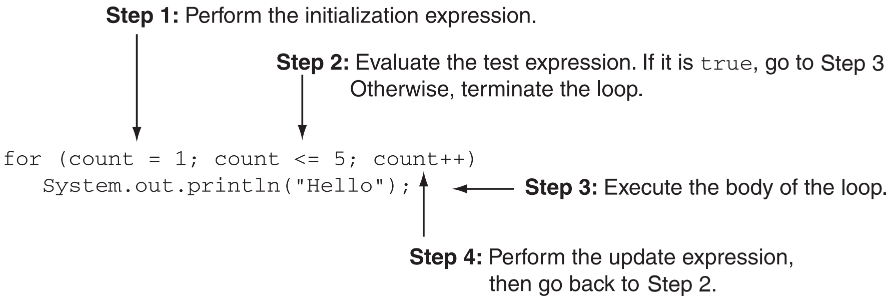
The first line of the `for` loop is known as the *loop header*. After the key word `for`, there are three expressions inside the parentheses, separated by semicolons. (Notice there is not a semicolon after the third expression.) The first expression is the *initialization expression*. It is normally used to initialize a control variable to its starting value. This is the first action performed by the loop, and it is done only once. The second expression is the *test expression*. This is a `boolean` expression that controls the execution of the loop. As long as this expression is `true`, the body of the `for` loop will repeat. The `for` loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the *update expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's control variable.

Here is an example of a simple `for` loop that prints “Hello” five times:

```
for (count = 1; count <= 5; count++)
   System.out.println("Hello");
```

In this loop, the initialization expression is `count = 1`, the test expression is `count <= 5`, and the update expression is `count++`. The body of the loop has one statement, which is the `println` statement. Figure 4-7 illustrates the sequence of events that takes place during the loop’s execution. Notice that Steps 2 through 4 are repeated as long as the test expression is `true`.

**Figure 4-7** Sequence of events in the `for` loop



**Step 1:** Perform the initialization expression.

**Step 2:** Evaluate the test expression. If it is `true`, go to Step 3 Otherwise, terminate the loop.

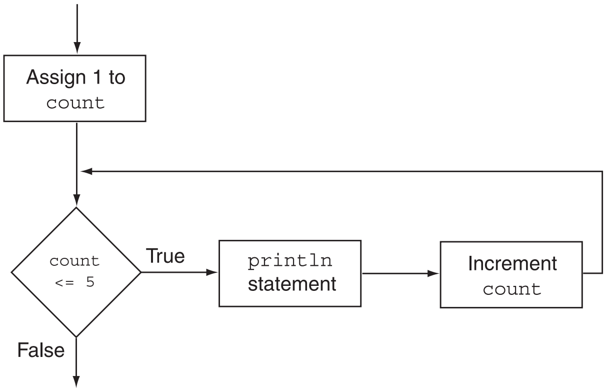
```
for (count = 1; count <= 5; count++)
   System.out.println("Hello");
```

**Step 3:** Execute the body of the loop.

**Step 4:** Perform the update expression, then go back to Step 2.

Figure 4-8 shows the loop’s logic in the form of a flowchart.

**Figure 4-8** Logic of the `for` loop



Notice how the control variable, `count`, is used to control the number of times that the loop iterates. During the execution of the loop, this variable takes on the values 1 through 5, and when the test expression `count <= 5` is `false`, the loop terminates. Because this variable keeps a count of the number of iterations, it is often called a *counter variable*.

Also notice that in this example the `count` variable is used only in the loop header, to control the number of loop iterations. It is not used for any other purpose. It is also possible to use the control variable within the body of the loop. For example, look at the following code:

```
for (number = 1; number <= 10; number++)
   System.out.print(number + " ");
```

The control variable in this loop is `number`. In addition to controlling the number of iterations, it is also used in the body of the loop. This loop will produce the following output:

```
1 2 3 4 5 6 7 8 9 10
```

As you can see, the loop displays the contents of the `number` variable during each iteration. The program in Code Listing 4-7 shows another example of a `for` loop that uses its control variable within the body of the loop. This program displays a table showing the numbers 1 through 10 and their squares.

**Code Listing 4-7** (`Squares.java`)

```java
/**
   This program demonstrates the for loop.
*/

public class Squares
{
   public static void main(String[] args)
   {
      int number; // Loop control variable

      System.out.println("Number Number Squared");
      System.out.println("------------------------");

      for (number = 1; number <= 10; number++)
      {
         System.out.println(number + "\t\t" +
                            number * number);
      }
   }
}
```

**Program Output**

```
Number      Number Squared
--------------------------
1               1
2               4
3               9
4               16
5               25
6               36
7               49
8               64
9               81
10              100
```

Figure 4-9 illustrates the sequence of events performed by this `for` loop.

**Figure 4-9** Sequence of events with the `for` loop in Code Listing 4-7

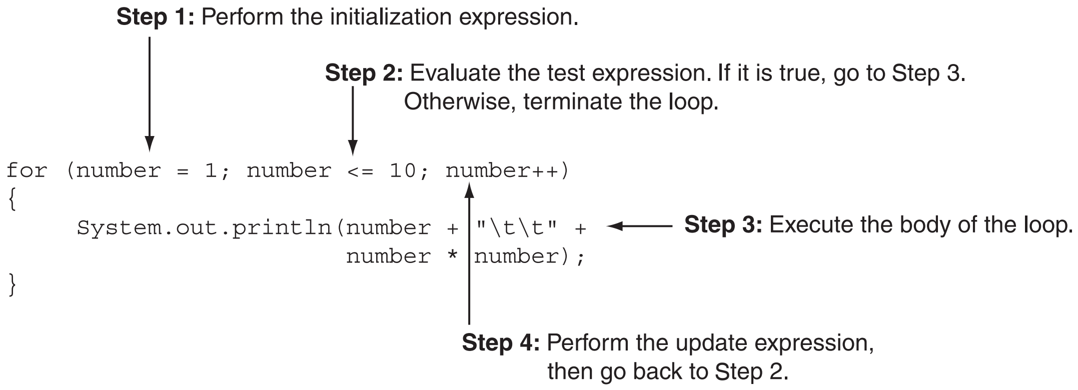
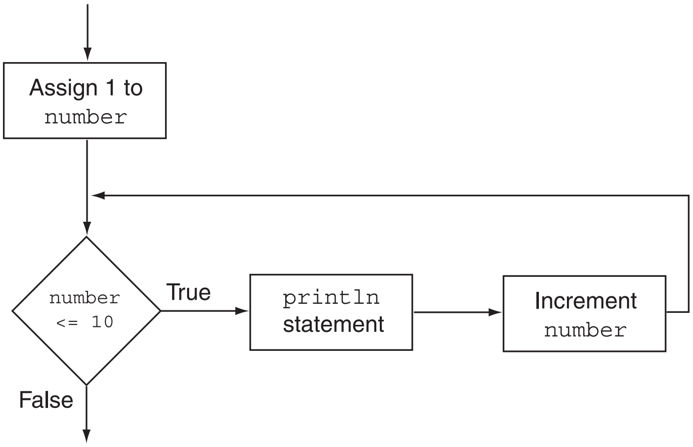

**Step 1:** Perform the initialization expression.
**Step 2:** Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop.
for (number = 1; number <= 10; number++)
{
System.out.println(number + "\t\t" +
number * number);
}
**Step 3:** Execute the body of the loop.
**Step 4:** Perform the update expression, then go back to Step 2.

Figure 4-10 shows the logic of the loop.

**Figure 4-10** Logic of the `for` loop in Code Listing 4-7


Assign 1 to number
number <= 10
True
False
println statement
Increment number

## The `for` Loop Is a Pretest Loop

Because the `for` loop tests its `boolean` expression before it performs an iteration, it is a pretest loop. It is possible to write a `for` loop in such a way that it will never iterate. Here is an example:

```
for (count = 11; count <= 10; count++)
   System.out.println("Hello");
```

Because the variable `count` is initialized to a value that makes the `boolean` expression `false` from the beginning, this loop terminates as soon as it begins.

## Avoid Modifying the Control Variable in the Body of the `for` Loop

Be careful not to place a statement that modifies the control variable in the body of the `for` loop. All modifications of the control variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the control variable, the loop probably will not terminate when you expect it to. The following loop, for example, increments `x` twice for each iteration:

```
for (x = 1; x <= 10; x++)
{
   System.out.println(x);
   x++; // Wrong!
}
```

## Other Forms of the Update Expression

You are not limited to using increment statements in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (number = 2; number <= 100; number += 2)
   System.out.println(number);
```

And here is a loop that counts backward from 10 to 0:

```
for (number = 10; number >= 0; number--)
   System.out.println(number);
```

## Declaring a Variable in the `for` Loop's Initialization Expression

Not only may the control variable be initialized in the initialization expression, but also it may be declared there. The following code shows an example. The following is a modified version of the loop in Code Listing 4-7:

```
for (int number = 1; number <= 10; number++)
{
   System.out.println(number + "\t\t" +
                      number * number);
}
```

In this loop, the `number` variable is both declared and initialized in the initialization expression. If the control variable is used only in the loop, it makes sense to declare it in the loop header. This makes the variable's purpose clearer.

When a variable is declared in the initialization expression of a `for` loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following program segment will not compile because the last `println` statement cannot access the variable `count`:

```
for (int count = 1; count <= 10; count++)
   System.out.println(count);
System.out.println("count is now " + count); // ERROR!
```

## Creating a User Controlled `for` Loop

Sometimes you want the user to determine the maximum value of the control variable in a `for` loop, and therefore determine the number of times the loop iterates. For example, look at the program in Code Listing 4-8. It is a modification of Code Listing 4-7. Instead of displaying the numbers 1 through 10 and their squares, this program allows the user to enter the maximum value to display.

**Code Listing 4-8** (`UserSquares.java`)

```
import java.util.Scanner;

/**
   This program demonstrates a user controlled for loop.
*/

public class UserSquares
{
   public static void main(String[] args)
   {
      int number;      // Loop control variable
      int maxValue;    // Maximum value to display

      System.out.println("I will display a table of " +
                         "numbers and their squares.");

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the maximum value to display.
      System.out.print("How high should I go? ");
      maxValue = keyboard.nextInt();

      // Display the table.
      System.out.println("Number Number Squared");
      System.out.println("-----------------------");
      for (number = 1; number <= maxValue; number++)
      {
         System.out.println(number + "\t\t" +
                            number * number);
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
I will display a table of numbers and their squares.
How high should I go? 7 [Enter]
Number    Number Squared
-----------------------------
1             1
2             4
3             9
4             16
5             25
6             36
7             49
```

In lines 21 and 22, which are before the loop, this program asks the user to enter the highest value to display. This value is stored in the `maxValue` variable as follows:

```
System.out.print("How high should I go? ");
maxValue = keyboard.nextInt();
```

In line 27, the for loop's test expression uses the value in the `maxValue` variable as the upper limit for the control variable as follows:

```
for (number = 1; number <= maxValue; number++)
```

In this loop, the `number` variable takes on the values 1 through `maxValue`, and then the loop terminates.

## Using Multiple Statements in the Initialization and Update Expressions

It is possible to execute more than one statement in the initialization expression and the update expression. When using multiple statements in either of these expressions, simply separate the statements with commas. For example, look at the loop in the following code, which has two statements in the initialization expression:

```
int x, y;
for (x = 1, y = 1; x <= 5; x++)
{
   System.out.println(x + " plus " + y +
                      " equals " + (x + y));
}
```

This loop's initialization expression is as follows:

```
x = 1, y = 1
```

This initializes two variables, `x` and `y`. The output produced by this loop is as follows:

```
1 plus 1 equals 2
2 plus 1 equals 3
3 plus 1 equals 4
```

```
4 plus 1 equals 5
5 plus 1 equals 6
```

We can further modify the loop to execute two statements in the update expression. Here is an example:

```
int x, y;
for (x = 1, y = 1; x <= 5; x++, y++)
{
   System.out.println(x + " plus " + y +
                      " equals " + (x + y));
}
```

The loop's update expression is as follows:

```
x++, y++
```

This update expression increments both the `x` and `y` variables. The output produced by this loop is as follows:

```
1 plus 1 equals 2
2 plus 2 equals 4
3 plus 3 equals 6
4 plus 4 equals 8
5 plus 5 equals 10
```

Connecting multiple statements with commas works well in the initialization and update expressions, but don't try to connect multiple `boolean` expressions this way in the test expression. If you wish to combine multiple `boolean` expressions in the test expression, you must use the `&&` or `||` operators.

## In the Spotlight:

### Designing a Count-Controlled `for` Loop

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour (KPH). She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour (MPH). The formula for converting KPH to MPH is

$$MPH = KPH * 0.6214$$

In the formula, $MPH$ is the speed in miles per hour and $KPH$ is the speed in kilometers per hour.

The table that your program displays should show speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

| KPH | MPH |
|---|---|
| 60 | 37.3 |
| 70 | 43.5 |
| 80 | 49.7 |
| *etc.* | |
| 130 | 80.8 |

After thinking about this table of values, you decide that you will write a `for` loop that uses a counter variable to hold the KPH speeds. The counter's starting value will be 60, its ending value will be 130, and you will add 10 to the counter variable after each iteration. Inside the loop you will use the counter variable to calculate a speed in MPH. Code Listing 4-9 shows the code.

**Code Listing 4-9** (SpeedConverter.java)

```
/**
   This program displays a table of speeds in
   kph converted to mph.
*/

public class SpeedConverter
{
  public static void main(String[] args)
  {
     // Constants
     final int STARTING_KPH = 60;       // Starting speed
     final int MAX_KPH = 130;           // Maximum speed
     final int INCREMENT = 10;          // Speed increment

     // Variables
     int kph;        // To hold the speed in kph
     double mph;     // To hold the speed in mph

     // Display the table headings.
     System.out.println("KPH\t\tMPH");
     System.out.println("-------------------");

     // Display the speeds.
     for (kph = STARTING_KPH; kph <= MAX_KPH; kph += INCREMENT)
     {
         // Calculate the mph.
         mph = kph * 0.6214;

         // Display the speeds in kph and mph.
         System.out.printf("%d\t\t%.1f\n", kph, mph);
     }
  }
}
```

**Program Output**

```
KPH     MPH
--------------
60      37.3
70      43.5
80      49.7
90      55.9
100     62.1
110     68.4
120     74.6
130     80.8
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

4.7 Name the three expressions that appear inside the parentheses in the `for` loop's header.

4.8 You want to write a `for` loop that displays "I love to program" 50 times. Assume that you will use a control variable named `count`.
   a) What initialization expression will you use?
   b) What test expression will you use?
   c) What update expression will you use?
   d) Write the loop.

4.9 What will the following program segments display?

**a)**
```
for (int count = 0; count < 6; count++)
   System.out.println(count + count);
```

**b)**
```
for (int value = -5; value < 5; value++)
   System.out.println(value);
```

**c)**
```
int x;

for (x = 5; x <= 14; x += 3)
   System.out.println(x);
System.out.println(x);
```

4.10 Write a `for` loop that displays your name 10 times.

4.11 Write a `for` loop that displays all of the odd numbers, 1 through 49.

4.12 Write a `for` loop that displays every fifth number, zero through 100.

## 4.6 Running Totals and Sentinel Values

**CONCEPT: A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator. A sentinel is a value that signals when the end of a list of values has been reached.**

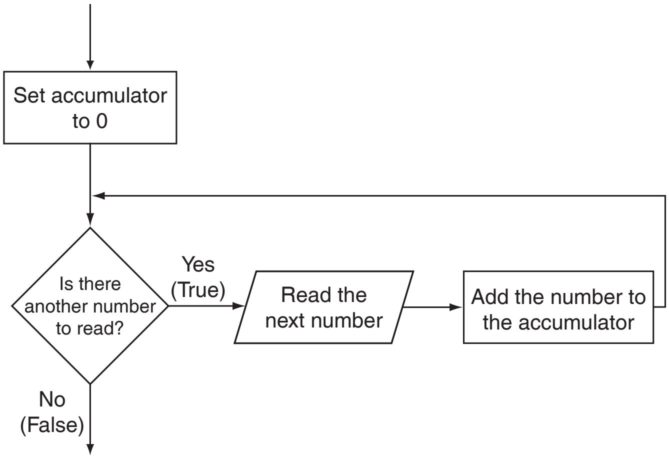
Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable used to accumulate the total of the numbers is called an *accumulator*. It is often said that the loop keeps a *running total* because it accumulates the total as it reads each number in the series. Figure 4-11 shows the general logic of a loop that calculates a running total.

**Figure 4-11** Logic for calculating a running total


Set accumulator to 0
Is there another number to read?
Yes (True)
Read the next number
Add the number to the accumulator
No (False)

When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

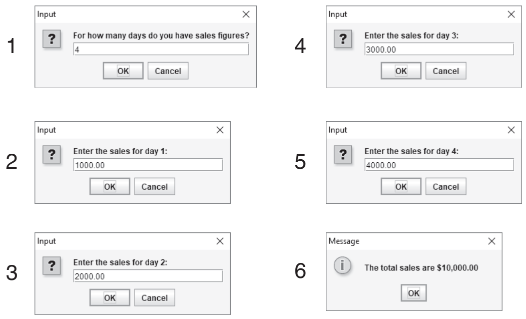
Let's look at a program that calculates a running total. Code Listing 4-10 calculates a company's total sales over a period of time by taking daily sales figures as input and calculating a running total of them as they are gathered. Figure 4-12 shows an example of interaction with the program.

**Code Listing 4-10** `(TotalSales.java)`

```
import javax.swing.JOptionPane;

/**
   This program calculates a running total.
*/

public class TotalSales
```

```java
{
   public static void main(String[] args)
   {
      int days;             // The number of days
      double sales;         // A day's sales figure
      double totalSales;    // Accumulator
      String input;         // To hold the user's input

      // Get the number of days.
      input = JOptionPane.showInputDialog("For how many days " +
                                   "do you have sales figures?");
      days = Integer.parseInt(input);

      // Set the accumulator to 0.
      totalSales = 0.0;

      // Get the sales figures and calculate a running total.
      for (int count = 1; count <= days; count++)
      {
         input = JOptionPane.showInputDialog("Enter the sales " +
                                       "for day " + count + ": ");
         sales = Double.parseDouble(input);
         totalSales += sales;  // Add sales to totalSales.
      }

      // Display the total sales.
      JOptionPane.showMessageDialog(null,
         String.format("The total sales are $%,.2f", totalSales));

      System.exit(0);
}
}
```

**Figure 4-12** Interaction with the `TotalSales` program (Oracle Corporate Counsel)



1 Input: For how many days do you have sales figures? 4

2 Input: Enter the sales for day 1: 1000.00

3 Input: Enter the sales for day 2: 2000.00

4 Input: Enter the sales for day 3: 3000.00

5 Input: Enter the sales for day 4: 4000.00

6 Message: The total sales are $10,000.00

Let's take a closer look at this program. In lines 17 and 18 the user is asked to enter the number of days for which he or she has sales figures. The number is read from an input dialog box and assigned to the `days` variable. Then, in line 22 the `totalSales` variable is assigned 0.0. In general programming terms, the `totalSales` variable is referred to as an accumulator. An *accumulator* is a variable that is initialized with a starting value, which is usually zero, and then accumulates a sum of numbers by having the numbers added to it. As you will see, it is critical that the accumulator is set to zero before values are added to it.

Next, the `for` loop in lines 25 through 31 executes. During each iteration of the loop, the user enters the amount of sales for a specific day, which are assigned to the `sales` variable. This is done in lines 27 through 29. Then, in line 30 the contents of `sales` is added to the existing value in the `totalSales` variable. (Note that line 30 does not assign `sales` to `totalSales`, but adds `sales` to `totalSales`. Put another way, this line increases `totalSales` by the amount in `sales`.)

Because `totalSales` was initially assigned 0.0, after the first iteration of the loop, `totalSales` will be set to the same value as `sales`. After each subsequent iteration, `totalSales` will be increased by the amount in `sales`. After the loop has finished, `totalSales` will contain the total of all the daily sales figures entered. Now it should be clear why we assigned 0.0 to `totalSales` before the loop executed. If `totalSales` started at any other value, the total would be incorrect.

## Using a Sentinel Value

The program in Code Listing 4-10 requires the user to know in advance the number of days for which he or she has sales figures. Sometimes the user has a very long list of input values, and doesn't know the exact number of items. In other cases, the user might be entering values from several lists and it is impractical to require that every item in every list is counted.

A technique that can be used in these situations is to ask the user to enter a sentinel value at the end of the list. A *sentinel value* is a special value that cannot be mistaken as a member of the list, and signals that there are no more values to be entered. When the user enters the sentinel value, the loop terminates.

The program in Code Listing 4-11 shows an example. It calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, and then −1 to signal the end of the list.

**Code Listing 4-11** (`SoccerPoints.java`)

```
import java.util.Scanner;

/**
   This program calculates the total number of points a
   soccer team has earned over a series of games. The user
   enters a series of point values, then -1 when finished.
*/

public class SoccerPoints
```

```
{
   public static void main(String[] args)
   {
      int points;                // Game points
      int totalPoints = 0;       // Accumulator initialized to 0

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Display general instructions.
      System.out.println("Enter the number of points your team");
      System.out.println("has earned for each game this season.");
      System.out.println("Enter -1 when finished.");
      System.out.println();

      // Get the first number of points.
      System.out.print("Enter game points or -1 to end: ");
      points = keyboard.nextInt();

      // Accumulate the points until -1 is entered.
      while (points != -1)
      {
         // Add points to totalPoints.
         totalPoints += points;

         // Get the next number of points.
         System.out.print("Enter game points or -1 to end: ");
         points = keyboard.nextInt();
      }

      // Display the total number of points.
      System.out.println("The total points are " +
                         totalPoints);
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the number of points your team
has earned for each game this season.
Enter -1 when finished.

Enter game points or -1 to end: 7 [Enter]
Enter game points or -1 to end: 9 [Enter]
Enter game points or -1 to end: 4 [Enter]
Enter game points or -1 to end: 6 [Enter]
Enter game points or -1 to end: 8 [Enter]
Enter game points or -1 to end: -1 [Enter]
The total points are 34
```

The value −1 was chosen for the sentinel because it is not possible for a team to score negative points. Notice that this program performs a priming read to get the first value. This makes it possible for the loop to terminate immediately if the user enters −1 as the first value. Also note that the sentinel value is not included in the running total.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

4.13 Write a `for` loop that repeats seven times, asking the user to enter a number. The loop should also calculate the sum of the numbers entered.

4.14 In the following program segment, which variable is the loop control variable (also known as the counter variable) and which is the accumulator?

```
int a, x = 0, y = 0;
while (x < 10)
{
   a = x * 2;
   y += a;
   x++;
}
System.out.println("The sum is " + y);
```

4.15 Why should you be careful when choosing a sentinel value?

## 4.7 Nested Loops

**CONCEPT: A loop that is inside another loop is called a nested loop.**

Nested loops are necessary when a task performs a repetitive operation and that task itself must be repeated. A clock is a good example of something that works like a nested loop. The program in Code Listing 4-12 uses nested loops to simulate a clock.

**Code Listing 4-12** (`Clock.java`)

```
/**
   This program uses nested loops to simulate a clock.
*/

public class Clock
{
   public static void main(String[] args)
   {
      // Simulate the clock.
      for (int hours = 1; hours <= 12; hours++)
      {
         for (int minutes = 0; minutes <= 59; minutes++)
         {
            for (int seconds = 0; seconds <= 59; seconds++)
15           {
16             System.out.printf("%02d:%02d:%02d\n", hours, minutes, seconds);
17           }
18         }
19       }
20     }
21   }
```

**Program Output**

```
01:00:00
01:00:01
01:00:02
01:00:03
```

*(The loop continues to count ... )*

```
12:59:57
12:59:58
12:59:59
```

The innermost loop (which begins at line 14) will iterate 60 times for each single iteration of the middle loop. The middle loop (which begins at line 12) will iterate 60 times for each single iteration of the outermost loop. When the outermost loop (which begins at line 10) has iterated 12 times, the middle loop will have iterated 720 times and the innermost loop will have iterated 43,200 times.

The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations before outer loops do.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

The program in Code Listing 4-13 shows another example. It is a program that a teacher might use to get the average of each student's test scores. In line 22 the user enters the number of students, and in line 26 the user enters the number of test scores per student. The `for` loop that begins in line 29 iterates once for each student. The nested inner `for` loop, in lines 36 through 41, iterates once for each test score.

**Code Listing 4-13** `(TestAverage2.java)`

```
1  import java.util.Scanner;
2
3  /**
4     This program demonstrates a nested loop.
5  */
6
7  public class TestAverage2
```

```
{
  public static void main(String [] args)
  {
     int numStudents,    // Number of students
         numTests,       // Number of tests per student
         score,          // Test score
         total;          // Accumulator for test scores
     double average;     // Average test score

    // Create a Scanner object for keyboard input.
    Scanner keyboard = new Scanner(System.in);

    // Get the number of students.
    System.out.print("How many students do you have? ");
    numStudents = keyboard.nextInt();

    // Get the number of test scores per student.
    System.out.print("How many test scores per student? ");
    numTests = keyboard.nextInt();

    // Process all the students.
    for (int student = 1; student <= numStudents; student++)
    {
        total = 0; // Set the accumulator to zero.

      // Get the test scores for a student.
      System.out.println("Student number " + student);
      System.out.println("--------------------");
      for (int test = 1; test <= numTests; test++)
      {
         System.out.print("Enter score " + test + ": ");
         score = keyboard.nextInt();
         total += score; // Add score to total.
      }

      // Calculate and display the average.
      average = total / numTests;
      System.out.printf("The average for student %d is %.1f.\n\n",
                        student, average);
    }
  }
}
```

**Program Output with Example Input Shown in Bold**

```
How many students do you have? 3 [Enter]
How many test scores per student? 3 [Enter]
```

```
Student number 1
-----------------------
Enter score 1: 100 [Enter]
Enter score 2: 95 [Enter]
Enter score 3: 90 [Enter]
The average for student number 1 is 95.0.

Student number 2
----------------------
Enter score 1: 80 [Enter]
Enter score 2: 81 [Enter]
Enter score 3: 82 [Enter]
The average for student number 2 is 81.0.

Student number 3
----------------------
Enter score 1: 75 [Enter]
Enter score 2: 85 [Enter]
Enter score 3: 80 [Enter]
The average for student number 3 is 80.0.
```

## In the Spotlight:

### Using Nested Loops to Print Patterns

One interesting way to learn about nested loops is to use them to display patterns on the screen. Let's look at a simple example. Suppose we want to print asterisks on the screen in the following rectangular pattern:

```
******
******
******
******
******
******
******
******
```

If you think of this pattern as having rows and columns, you can see that it has eight rows, and each row has six columns. The following code can be used to display one row of asterisks:

```
final int COLS = 6;
for (int col = 0; col < COLS; col++)
{
   System.out.print("*");
}
```

If we run this code in a program, it will produce the following output:

```
******
```

To complete the entire pattern, we need to execute this loop eight times. We can place the loop inside another loop that iterates eight times, as shown here:

```
1    final int COLS = 6;
2    final int ROWS = 8;
3    for (int row = 0; row < ROWS; row++)
4    {
5       for (int col = 0; col < COLS; col++)
6       {
7          System.out.print("*");
8       }
9       System.out.println();
10   }
```

The outer loop iterates eight times. Each time it iterates, the inner loop iterates six times. (Notice that in line 9, after each row has been printed, we call the `System.out.println()` method. We have to do that to advance the screen cursor to the next line at the end of each row. Without that statement, all the asterisks will be printed in one long row on the screen.)

We could easily write a program that prompts the user for the number of rows and columns, as shown in Code Listing 4-14.

**Code Listing 4-14** (`RectangularPattern.java`)

```
import java.util.Scanner;

/**
   This program displays a rectangular pattern
   of asterisks.
*/

public class RectangularPattern
{
   public static void main(String[] args)
   {
      int rows, cols;

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of rows and columns.
      System.out.print("How many rows? ");
      rows = keyboard.nextInt();
      System.out.print("How many columns? ");
      cols = keyboard.nextInt();

      for (int r = 0; r < rows; r++)
      {
         for (int c = 0; c < cols; c++)
         {
            System.out.print("*");
      }
      System.out.println();
    }
  }
}
```

**Program Output with Example Input Shown in Bold**

```
How many rows? 5 [Enter]
How many columns?  10 [Enter]
**********
**********
**********
**********
**********
```

Let's look at another example. Suppose you want to print asterisks in a pattern that looks like the following triangle:

```
*
**
***
****
*****
******
*******
********
```

Once again, think of the pattern as being arranged in rows and columns. The pattern has a total of eight rows. In the first row, there is one column. In the second row, there are two columns. In the third row, there are three columns. This continues to the eighth row, which has eight columns. Code Listing 4-15 shows the program that produces this pattern.

**Code Listing 4-15** `(TrianglePattern.java)`

```
import java.util.Scanner;

/**
   This program displays a triangle pattern.
*/

public class TrianglePattern
{
   public static void main(String[] args)
   {
      final int BASE_SIZE = 8;

      for (int r = 0; r < BASE_SIZE; r++)
14        {
15           for (int c = 0; c < (r + 1); c++)
16           {
17              System.out.print("*");
18           }
19           System.out.println();
20        }
21     }
22  }
```

**Program Output**

```
*
**
***
****
*****
******
*******
********
```

The outer loop (which begins in line 13) will iterate eight times. As the loop iterates, the variable `r` will be assigned the values 0 through 7.

For each iteration of the outer loop, the inner loop will iterate `r + 1` times. So,

- During the outer loop's first iteration, the variable `r` is assigned 0. The inner loop iterates one time, printing one asterisk.
- During the outer loop's second iteration, the variable `r` is assigned 1. The inner loop iterates two times, printing two asterisks.
- During the outer loop's third iteration, the variable `r` is assigned 2. The inner loop iterates three times, printing three asterisks.
- And so forth.

Let's look at another example. Suppose you want to display the following stair-step pattern:

```
#
 #
  #
   #
    #
     #
```

The pattern has six rows. In general, we can describe each row as having some number of spaces followed by a # character. Here's a row-by-row description:

First row: 0 spaces followed by a # character.
Second row: 1 space followed by a # character.
Third row: 2 spaces followed by a # character.
Fourth row: 3 spaces followed by a # character.
Fifth row: 4 spaces followed by a # character.
Sixth row: 5 spaces followed by a # character.

To display this pattern, we can write code containing a pair of nested loops that work in the following manner:

- The outer loop will iterate six times. Each iteration will perform the following:
  - The inner loop will display the correct number of spaces, side by side.
  - Then, a # character will be displayed.

Code Listing 4-16 shows the Java code.

**Code Listing 4-16** `(StairStepPattern.java)`

```
 1  import java.util.Scanner;
 2
 3  /**
 4    This program displays a stairstep pattern.
 5  */
 6
 7  public class StairStepPattern
 8  {
 9     public static void main(String[] args)
10     {
11        final int NUM_STEPS = 6;
12
13        for (int r = 0; r < NUM_STEPS; r++)
14        {
15           for (int c = 0; c < r; c++)
16           {
17              System.out.print(" ");
18           }
19           System.out.println("#");
20        }
21     }
22  }
```

**Program Output**

```
#
 #
  #
   #
    #
     #
```

The outer loop (which begins in line 13) will iterate six times. As the loop iterates, the variable `r` will be assigned the values 0 through 5.

For each iteration of the outer loop, the inner loop will iterate `r` times. So,

- During the outer loop's first iteration, the variable `r` is assigned 0. The inner loop will not execute at this time.

- During the outer loop's second iteration, the variable `r` is assigned 1. The inner loop iterates one time, printing one space.
- During the outer loop's third iteration, the variable `r` is assigned 2. The inner loop iterates two times, printing two spaces.
- And so forth.

## 4.8 The break and continue Statements (Optional)

**CONCEPT: The `break` statement causes a loop to terminate early. The `continue` statement causes a loop to stop its current iteration and begin the next one.**

The `break` statement, which was used with the `switch` statement in Chapter 3, can also be placed inside a loop. When it is encountered, the loop stops and the program jumps to the statement immediately following the loop. Although it is perfectly acceptable to use the `break` statement in a `switch` statement, it is considered taboo to use it in a loop. This is because it bypasses the normal condition that is required to terminate the loop, and it makes code difficult to understand and debug. For this reason, you should avoid using the `break` statement in a loop when possible.

The `continue` statement causes the current iteration of a loop to end immediately. When `continue` is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration. In a `while` loop, this means the program jumps to the `boolean` expression at the top of the loop. As usual, if the expression is still `true`, the next iteration begins. In a `do-while` loop, the program jumps to the `boolean` expression at the bottom of the loop, which determines whether the next iteration will begin. In a `for` loop, `continue` causes the update expression to be executed, and then the test expression is evaluated.

The `continue` statement should also be avoided. Like the `break` statement, it bypasses the loop's logic and makes the code difficult to understand and debug.

## 4.9 Deciding Which Loop to Use

**CONCEPT: Although most repetitive algorithms can be written with any of the three types of loops, each works best in different situations.**

Each of Java's three loops is ideal to use in different situations. The following is a short summary of when each loop should be used:

- **The `while` loop.** The `while` loop is a pretest loop. It is ideal in situations where you do not want the loop to iterate if the condition is `false` from the beginning. It is also ideal if you want to use a sentinel value to terminate the loop.
- **The `do-while` loop.** The `do-while` loop is a posttest loop. It is ideal in situations where you always want the loop to iterate at least once.
- **The `for` loop.** The `for` loop is a pretest loop that has built-in expressions for initializing, testing, and updating. These expressions make it very convenient to use a loop control variable as a counter. The `for` loop is ideal in situations where the exact number of iterations is known.

## 4.10 Introduction to File Input and Output

**CONCEPT: The Java API provides several classes that you can use for writing data to a file and reading data from a file. To write data to a file, you can use the `PrintWriter` class and, optionally, the `FileWriter` class. To read data from a file, you can use the `Scanner` class and the `File` class.**

The programs you have written so far require you to reenter data each time the program runs. This is because the data stored in variables and objects in RAM disappears once the program stops running. To retain data between the times it runs, a program must have a way of saving the data.

Data may be saved in a *file*, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. The data can then be retrieved and used at a later time. In general, there are three steps that are taken when a file is used by a program:

1. The file must be *opened*. When the file is opened, a connection is created between the file and the program.
2. Data is then written to the file or read from the file.
3. When the program is finished using the file, the file must be *closed*.

In this section we will discuss how to write Java programs that write data to files and read data from files. The terms *input file* and *output file* are commonly used. An *input file* is a file that a program reads data from. It is called an input file because the data stored in it serves as input to the program. An *output file* is a file that a program writes data to. It is called an output file because the program stores output in the file.

In general, there are two types of files: text and binary. A *text file* contains data that has been encoded as text, using a scheme such as Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A *binary file* contains data that has not been converted to text. As a consequence, you cannot view the contents of a binary file with a text editor. In this chapter, we will discuss how to work with text files. Binary files are discussed in Chapter 11.

The Java API provides a number of classes that you will use to work with files. To use these classes, you will place the following `import` statement near the top of your program:

```
import java.io.*;
```

### Using the `PrintWriter` Class to Write Data to a File

To write data to a file you will create an instance of the `PrintWriter` class. The `PrintWriter` class allows you to open a file for writing. It also allows you to write data to the file using the same `print` and `println` methods that you have been using to display data on the screen. You pass the name of the file that you wish to open, as a string, to the `PrintWriter` class's constructor. For example, the following statement creates a `PrintWriter` object and passes the file name *StudentData.txt* to the constructor.

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```

This statement will create an empty file named *StudentData.txt* and establish a connection between it and the `PrintWriter` object that is referenced by `outputFile`. The file will be created in the current directory or folder.

You may also pass a reference to a `String` object as an argument to the `PrintWriter` constructor. For example, in the following code the user specifies the name of the file.

```
String filename;
filename = JOptionPane.showInputDialog("Enter the filename.");
PrintWriter outputFile = new PrintWriter(filename);
```

**WARNING!** If the file that you are opening with the `PrintWriter` object already exists, it will be erased and an empty file by the same name will be created.

Once you have created an instance of the `PrintWriter` class and opened a file, you can write data to the file using the `print` and `println` methods. You already know how to use `print` and `println` with `System.out` to display data on the screen. They are used the same way with a `PrintWriter` object to write data to a file. For example, assuming that `outputFile` references a `PrintWriter` object, the following statement writes the string `"Jim"` to the file:

```
outputFile.println("Jim");
```

When the program is finished writing data to the file, it must close the file. To close the file use the `PrintWriter` class's `close` method. Here is an example of the method's use:

```
outputFile.close();
```

Your application should always close files when finished with them. This is because the system creates one or more buffers when a file is opened. A *buffer* is a small "holding section" of memory. When a program writes data to a file, that data is first written to the buffer. When the buffer is filled, all the information stored there is written to the file. This technique increases the system's performance because writing data to memory is faster than writing it to a disk. The `close` method writes any unsaved data remaining in the file buffer.

Once a file is closed, the connection between it and the `PrintWriter` object is removed. In order to perform further operations on the file, it must be opened again.

### More about the `PrintWriter` Class's `println` Method

The `PrintWriter` class's `println` method writes a line of data to a file. For example, assume an application creates a file and writes three students' first names and their test scores to the file with the following code:

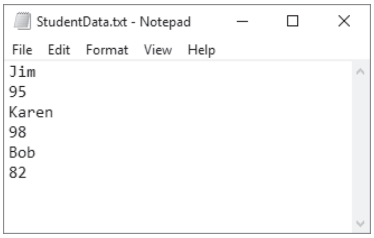
```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
outputFile.println("Jim");
outputFile.println(95);
outputFile.println("Karen");
outputFile.println(98);
outputFile.println("Bob");
outputFile.println(82);
outputFile.close();
```

The `println` method writes data to the file and then writes a newline character immediately after the data. You can visualize the data written to the file in the following manner:

```
Jim<newline>95<newline>Karen<newline>98<newline>Bob<newline>82<newline>
```

The newline characters are represented here as <*newline*>. You do not actually see the newline characters, but when the file is opened in a text editor such as Notepad, its contents will appear as shown in Figure 4-13. As you can see from the figure, each newline character causes the data that follows it to be displayed on a new line.

**Figure 4-13** File contents displayed in Notepad (Microsoft Corporation)



```
Jim
95
Karen
98
Bob
82
```

In addition to separating the contents of a file into lines, the newline character also serves as a delimiter. A *delimiter* is an item that separates other items. When you write data to a file using the `println` method, newline characters will separate the individual items of data. Later you will see that the individual items of data in a file must be separated in order for them to be read from the file.

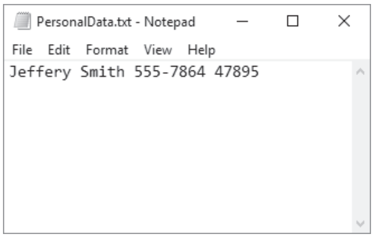
### The `PrintWriter` Class's `print` Method

The `print` method is used to write an item of data to a file without writing the newline character. For example, look at the following code:

```
String name = "Jeffrey Smith";
String phone = "555-7864";
int idNumber = 47895;
PrintWriter outputFile = new PrintWriter("PersonalData.txt");
outputFile.print(name + " ");
outputFile.print(phone + " ");
outputFile.println(idNumber);
outputFile.close();
```

This code uses the `print` method to write the contents of the `name` object to the file, followed by a space (`" "`). Then it uses the `print` method to write the contents of the `phone` object to the file, followed by a space. Then it uses the `println` method to write the contents of the `idNumber` variable, followed by a newline character. Figure 4-14 shows the contents of the file displayed in Notepad.

**Figure 4-14** Contents of file displayed in Notepad (Microsoft Corporation)



```
Jeffery Smith 555-7864 47895
```

### Adding a `throws` Clause to the Method Header

When an unexpected event occurs in a Java program, it is said that the program throws an exception. For now, you can think of an *exception* as a signal indicating that the program cannot continue until the unexpected event has been dealt with. For example, suppose you create a `PrintWriter` object and pass the name of a file to its constructor. The `PrintWriter` object attempts to create the file, but unexpectedly, the disk is full and the file cannot be created. Obviously the program cannot continue until this situation has been dealt with, so an exception is thrown, which causes the program to suspend normal execution.

When an exception is thrown, the method that is executing must either deal with the exception or throw it again. If the `main` method throws an exception, the program halts and an error message is displayed. Because `PrintWriter` objects are capable of throwing exceptions, we must either write code that deals with the possible exceptions, or allow our methods to rethrow the exceptions when they occur. In Chapter 11 you will learn all about exceptions and how to respond to them, but for now, we will simply allow our methods to rethrow any exceptions that might occur.

To allow a method to rethrow an exception that has not been dealt with, you simply write a `throws` clause in the method header. The `throws` clause must indicate the type of exception that might be thrown. The following is an example:

```
public static void main(String[] args) throws IOException
```

This header indicates that the `main` method is capable of throwing an exception of the `IOException` type. This is the type of exception that `PrintWriter` objects are capable of throwing. So, any method that uses `PrintWriter` objects and does not respond to their exceptions must have this `throws` clause listed in its header.

In addition, any method that calls a method that uses a `PrintWriter` object should have a `throws IOException` clause in its header. For example, suppose the `main` method does not perform any file operations, but calls a method named `buildFile` that opens a file and writes data to it. Both the `buildFile` and `main` methods should have a `throws IOException` clause in their headers. Otherwise a compiler error will occur.

### An Example Program

Let's look at an example program that writes data to a file. The program in Code Listing 4-17 writes the names of your friends to a file.

**Code Listing 4-17** (`FileWriteDemo.java`)

```
import java.util.Scanner;     // Needed for Scanner class
import java.io.*;             // Needed for File I/O classes

/**
  This program writes data to a file.
*/

public class FileWriteDemo
{
```

```java
public static void main(String[] args) throws IOException
{
   String filename;      // File name
   String friendName;    // Friend's name
   int numFriends;       // Number of friends

   // Create a Scanner object for keyboard input.
   Scanner keyboard = new Scanner(System.in);

   // Get the number of friends.
   System.out.print("How many friends do you have? ");
   numFriends = keyboard.nextInt();

   // Consume the remaining newline character.
   keyboard.nextLine();

   // Get the filename.
   System.out.print("Enter the filename: ");
   filename = keyboard.nextLine();

   // Open the file.
   PrintWriter outputFile = new PrintWriter(filename);

   // Get data and write it to the file.
   for (int i = 1; i <= numFriends; i++)
   {
       // Get the name of a friend.
       System.out.print("Enter the name of friend " +
                          "number " + i + ": ");
       friendName = keyboard.nextLine();

       // Write the name to the file.
       outputFile.println(friendName);
    }

    // Close the file.
    outputFile.close();
    System.out.println("Data written to the file.");
  }
}
```

### Program Output with Example Input Shown in Bold

```
How many friends do you have? 5 [Enter]
Enter the filename: MyFriends.txt [Enter]
Enter the name of friend number 1: Joe [Enter]
Enter the name of friend number 2: Rose [Enter]
Enter the name of friend number 3: Greg [Enter]
Enter the name of friend number 4: Kirk [Enter]
Enter the name of friend number 5: Renee [Enter]
Data written to the file.
```

The `import` statement in line 2 is necessary because this program uses the `PrintWriter` class. In addition, the `main` method header, in line 10, has a `throws IOException` clause because objects of the `PrintWriter` class can potentially throw an `IOException`.

This program asks the user to enter the number of friends he or she has (in lines 20 and 21), then a name for the file that will be created (in lines 27 and 28). The `filename` variable references the name of the file, and is used in the following statement, in line 31:

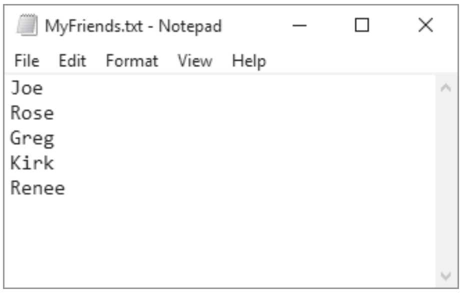
```
PrintWriter outputFile = new PrintWriter(filename);
```

This statement opens the file and creates a `PrintWriter` object that can be used to write data to the file. The `for` loop in lines 34 through 43 performs an iteration for each friend that the user has, each time asking for the name of a friend. The user's input is referenced by the `friendName` variable. Once the name is entered, it is written to the file with the following statement, which appears in line 42:

```
outputFile.println(friendName);
```

After the loop finishes, the file is closed in line 46. After the program is executed with the input shown in the example run, the file *MyFriends.txt* will be created. If we open the file in Notepad, we will see its contents as shown in Figure 4-15.

**Figure 4-15** Contents of the file displayed in Notepad (Microsoft Corporation)



```
Joe
Rose
Greg
Kirk
Renee
```

### Review

Before moving on, let's review the basic steps necessary when writing a program that writes data to a file:

1. You need the `import java.io.*;` statement in the top section of your program.
2. Because we have not yet learned how to respond to exceptions, any method that uses a `PrintWriter` object must have a `throws IOException` clause in its header.
3. You create a `PrintWriter` object and pass the name of the file as a string to the constructor.

4. You use the `PrintWriter` class's `print` and `println` methods to write data to the file.
5. When finished writing to the file, you use the `PrintWriter` class's `close` method to close the file.

## Appending Data to a File

When you pass the name of a file to the `PrintWriter` constructor, and the file already exists, it will be erased and a new empty file with the same name will be created. Sometimes, however, you want to preserve an existing file and append new data to its current contents. Appending to a file means writing new data to the end of the data that already exists in the file.

To append data to an existing file, you first create an instance of the `FileWriter` class. You pass two arguments to the `FileWriter` constructor: a string containing the name of the file, and the `boolean` value `true`. Here is an example:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
```

This statement creates a `FileWriter` object and opens the file *MyFriends.txt* for writing. Any data written to the file will be appended to the file's existing contents. (If the file does not exist, it will be created.)

You still need to create a `PrintWriter` object so you can use the `print` and `println` methods to write data to the file. When you create the `PrintWriter` object, you pass a reference to the `FileWriter` object as an argument to the `PrintWriter` constructor. For example, look at the following code:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
```

This creates a `PrintWriter` object that can be used to write data to the file *MyFriends.txt*. Any data that is written to the file will be appended to the file's existing contents. For example, assume the file *MyFriends.txt* exists and contains the following data:

```
Joe
Rose
Greg
Kirk
Renee
```

The following code opens the file and appends additional data to its existing contents:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
outputFile.println("Bill");
outputFile.println("Steven");
outputFile.println("Sharon");
outputFile.close();
```

After this code executes, the *MyFriends.txt* file will contain the following data:

```
Joe
Rose
Greg
Kirk
Renee
Bill
Steven
Sharon
```

> **NOTE:** The `FileWriter` class also throws an `IOException` if the file cannot be opened for any reason.

## Specifying the File Location

When you open a file you may specify its path along with its filename. On a Windows computer, paths contain backslash characters. Remember that when a single backslash character appears in a string literal, it marks the beginning of an escape sequence such as `"\n"`. Two backslash characters in a string literal represent a single backslash. So, when you provide a path in a string literal, and the path contains backslash characters, you must use two backslash characters in the place of each single backslash character.

For example, the path `"E:\\Names.txt"` specifies that *Names.txt* is in the root folder of drive E:, and the path `"C:\\MyData\\Data.txt"` specifies that *Data.txt* is in the *MyData* folder on drive C:. In the following statement, the file *Pricelist.txt* is created in the root folder of drive H:.

```
PrintWriter outputFile = new PrintWriter("H:\\PriceList.txt");
```

You only need to use double backslashes if the file's path is in a string literal. If your program asks the user to enter a path into a `String` object, which is then passed to the `PrintWriter` or `FileWriter` constructor, the user does not have to enter double backslashes.

> **TIP:** Java allows you to substitute forward slashes for backslashes in a Windows path. For example, the path `"C:\\MyData\\Data.txt"` could be written as `"C:/MyData/Data.txt"`. This eliminates the need to use double backslashes.

On a UNIX or Linux computer, you can provide a path without any modifications. Here is an example:

```
PrintWriter outputFile = new PrintWriter("/home/rharrison/names.txt");
```

## Reading Data from a File

In Chapter 2 you learned how to use the `Scanner` class to read input from the keyboard. To read keyboard input, recall that we create a `Scanner` object, passing `System.in` to the `Scanner` class constructor. Here is an example:

```
Scanner keyboard = new Scanner(System.in);
```

Recall that the `System.in` object represents the keyboard. Passing `System.in` as an argument to the `Scanner` constructor specifies that the keyboard is the `Scanner` object's source of input.

You can also use the `Scanner` class to read input from a file. Instead of passing `System.in` to the `Scanner` class constructor, you pass a reference to a `File` object. Here is an example:

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

The first statement creates an instance of the `File` class. The `File` class is in the Java API, and is used to represent a file. Notice that we have passed the string `"Customers.txt"` to the constructor. This creates a `File` object that represents the file *Customers.txt*.

In the second statement we pass a reference to this `File` object as an argument to the `Scanner` class constructor. This creates a `Scanner` object that uses the file *Customers.txt* as its source of input. You can then use the same `Scanner` class methods that you learned about in Chapter 2 to read items from the file. (See Table 2-17 for a list of commonly used methods.)

When you are finished reading from the file, you use the `Scanner` class's `close` method to close the file. For example, assuming the variable `inputFile` references a `Scanner` object, the following statement closes the file that is the object's source of input:

```
inputFile.close();
```

## Reading Lines from a File with the `nextLine` Method

The `Scanner` class's `nextLine` method reads a line of input, and returns the line as a `String`. The program in Code Listing 4-18 demonstrates how the `nextLine` method can be used to read a line from a file. This program asks the user to enter a filename. It then displays the first line in the file on the screen.

**Code Listing 4-18** `(ReadFirstLine.java)`

```
import java.util.Scanner; // Needed for Scanner class
import java.io.*;         // Needed for File and IOException

/**
   This program reads the first line from a file.
*/

public class ReadFirstLine
{
   public static void main(String[] args) throws IOException
   {
      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

15        // Get the file name.
16        System.out.print("Enter the name of a file: ");
17        String filename = keyboard.nextLine();
18
19        // Open the file.
20        File file = new File(filename);
21        Scanner inputFile = new Scanner(file);
22
23        // Read the first line from the file.
24        String line = inputFile.nextLine();
25
26        // Display the line.
27        System.out.println("The first line in the file is:");
28        System.out.println(line);
29
30        // Close the file.
31        inputFile.close();
32     }
33  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the name of a file: MyFriends.txt [Enter]
The first line in the file is:
Joe
```

This program gets the name of a file from the user in line 17. A `File` object is created in line 20 to represent the file, and a `Scanner` object is created in line 21 to read data from the file. Line 24 reads a line from the file. After this statement executes, the `line` variable references a `String` object holding the line that was read from the file. The line is displayed on the screen in line 28, and the file is closed in line 31.

It's worth pointing out that this program creates two separate `Scanner` objects. The `Scanner` object that is created in line 13 reads data from the keyboard, and the `Scanner` object that is created in line 21 reads data from a file.

When a file is opened for reading, a special value known as a *read position* is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. When a file is opened, its read position is set to the first item in the file. When the item is read, the read position is advanced to the next item in the file. As subsequent items are read, the internal read position advances through the file. For example, consider the file *Quotation.txt*, shown in Figure 4-16. As you can see from the figure, the file has three lines.

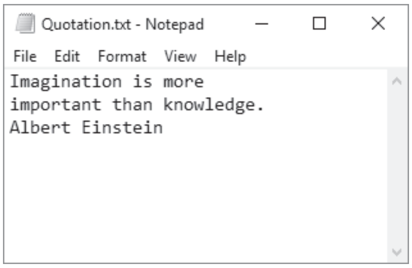
You can visualize that the data is stored in the file in the following manner:

```
Imagination is more<newline>important than knowledge.<newline>
Albert Einstein<newline>
```

Suppose a program opens the file with the following code:

```
File file = new File("Quotation.txt");
Scanner inputFile = new Scanner(file);
```

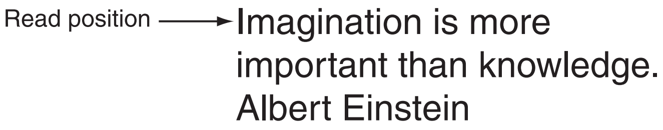
**Figure 4-16** File with three lines (Microsoft Corporation)



Imagination is more
important than knowledge.
Albert Einstein

When this code opens the file, its read position is at the beginning of the first line, as illustrated in Figure 4-17.

**Figure 4-17** Initial read position



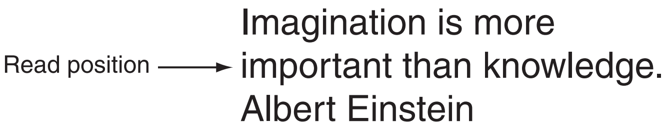
Imagination is more
important than knowledge.
Albert Einstein

Now, suppose the program uses the following statement to read a line from the file:

```
String str = inputFile.nextLine();
```

This statement will read a line from the file, beginning at the current read position. After the statement executes, the object referenced by `str` will contain the string “Imagination is more”. The file’s read position will be advanced to the next line, as illustrated in Figure 4-18.

**Figure 4-18** Read position after first line is read



Imagination is more
important than knowledge.
Albert Einstein

If the `nextLine` method is called again, the second line will be read from the file and the file’s read position will be advanced to the third line. After all the lines have been read, the read position will be at the end of the file.

**NOTE:** The string that is returned from the `nextLine` method will not contain the newline character.

## Adding a `throws` Clause to the Method Header

When you pass a `File` object reference to the `Scanner` class constructor, the constructor will throw an exception of the `IOException` type if the specified file is not found. So, you will need to write a throws `IOException` clause in the header of any method that passes a `File` object reference to the `Scanner` class constructor.

### Detecting the End of a File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, the *MyFriends.txt* file that was created by the program in Code Listing 4-17 can have any number of names stored in it. This is because the program asks the user for the number of friends that he or she has. If the user enters 5 for the number of friends, the program creates a file with five names in it. If the user enters 100, the program creates a file with 100 names in it.

The `Scanner` class has a method named `hasNext` that can be used to determine whether the file has more data that can be read. You call the `hasNext` method before you call any other methods to read from the file. If there is more data that can be read from the file, the `hasNext` method returns `true`. If the end of the file has been reached and there is no more data to read, the `hasNext` method returns `false`.

Code Listing 4-19 shows an example. The program reads the file containing the names of your friends, which was created by the program in Code Listing 4-17.

**Code Listing 4-19** (`FileReadDemo.java`)

```java
import java.util.Scanner;   // Needed for the Scanner class
import java.io.*;           // Needed for the File and IOException

/**
   This program reads data from a file.
*/

public class FileReadDemo
{
   public static void main(String[] args) throws IOException
   {
      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the filename.
      System.out.print("Enter the filename: ");
      String filename = keyboard.nextLine();
18
19       // Open the file.
20       File file = new File(filename);
21       Scanner inputFile = new Scanner(file);
22
23       // Read lines from the file until no more are left.
24       while (inputFile.hasNext())
25       {
26          // Read the next name.
27          String friendName = inputFile.nextLine();
28
29          // Display the last name read.
30          System.out.println(friendName);
31      }
32
33      // Close the file.
34      inputFile.close();
35    }
36 }
```

**Program Output with Example Input Shown in Bold**

```
Enter the filename: MyFriends.txt [Enter]
Joe
Rose
Greg
Kirk
Renee
```

The file is opened and a `Scanner` object to read it is created in line 21. The loop in lines 24 through 31 reads all of the lines from the file and displays them. In line 24 the loop calls the `Scanner` object's `hasNext` method. If the method returns `true`, then the file has more data to read. In that case, the next line is read from the file in line 27, and is displayed in line 30. The loop repeats until the `hasNext` method returns `false` in line 24. Figure 4-19 shows the logic of reading a file until the end is reached.

### Reading Primitive Values from a File

Recall from Chapter 2 that the `Scanner` class provides methods for reading primitive values. These methods are named `nextByte`, `nextDouble`, `nextFloat`, `nextInt`, nextLine, `nextLong`, and `nextShort`. Table 2-17 gives more information on each of these methods, which can be used to read primitive values from a file.

The program in Code Listing 4-20 demonstrates how the `nextDouble` method can be used to read floating-point values from a file. The program reads the contents of a file named *Numbers.txt*. The contents of the *Numbers.txt* file are shown in Figure 4-20. As you can see, the file contains a series of floating-point numbers. The program reads all of the numbers from the file and calculates their total.

**Figure 4-19** Logic of reading a file until the end is reached

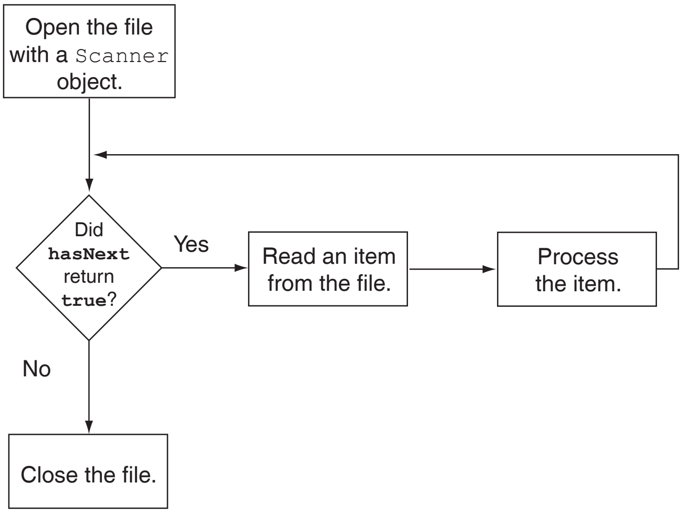
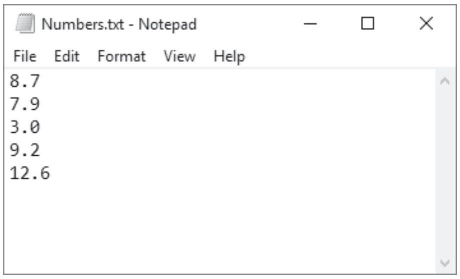

Open the file with a Scanner object.
Did hasNext return true?
Yes
Read an item from the file.
Process the item.
No
Close the file.

**Figure 4-20** Contents of *Numbers.txt* (Microsoft Corporation)


Numbers.txt - Notepad
File Edit Format View Help

```
8.7
7.9
3.0
9.2
12.6
```

**Code Listing 4-20** (FileSum.java)

```
import java.util.Scanner;
import java.io.*;

/**
   This program reads a series of numbers from a file and
   accumulates their sum.
*/

public class FileSum
{
   public static void main(String[] args) throws IOException
   {
      double sum = 0.0; // Accumulator, initialized to 0

      // Open the file for reading.
      File file = new File("Numbers.txt");
      Scanner inputFile = new Scanner(file);


      // Read all of the values from the file
      // and calculate their total.
      while (inputFile.hasNext())
      {
         // Read a value from the file.
         double number = inputFile.nextDouble();

         // Add the number to sum.
         sum = sum + number;
      }

      // Close the file.
      inputFile.close();

      // Display the sum of the numbers.
      System.out.println("The sum of the numbers in " +
                         "Numbers.txt is " + sum);
   }
}
```

**Program Output**

```
The sum of the numbers in Numbers.txt is 41.4
```

### Review

Let's quickly review the steps necessary when writing a program that reads data from a file:

1. You will need the import `java.util.Scanner;` statement in the top section of your program, so you can use the `Scanner` class. You will also need the `import java.io.*;` statement in the top section of your program. This is required by the `File` class.
2. Because we have not yet learned how to respond to exceptions, any method that uses a `Scanner` object to open a file must have a `throws IOException` clause in its header.
3. You create a `File` object and pass the name of the file as a string to the constructor.
4. You create a `Scanner` object and pass a reference to the `File` object as an argument to the constructor.
5. You use the `Scanner` class's `nextLine` method to read a line from the file. The method returns the line of data as a string. To read primitive values, use methods such as `nextInt`, `nextDouble`, and so forth.

6. Call the `Scanner` class's `hasNext` method to determine whether there is more data to read from the file. If the method returns `true`, then there is more data to read. If the method returns `false`, you have reached the end of the file.
7. When finished writing to the file, you use the `Scanner` class's `close` method to close the file.

## Checking for a File's Existence

It's usually a good idea to make sure that a file exists before you try to open it for input. If you attempt to open a file for input, and the file does not exist, the program will throw an exception and halt. For example, the program you saw in Code Listing 4-20 will throw an exception at line 17 if the file *Numbers.txt* does not exist. Here is an example of the error message that will be displayed when this happens:

```
Exception in thread "main" java.io.FileNotFoundException: Numbers.txt (The
system cannot find the file specified)
   at java.io.FileInputStream.open(Native Method)
   at java.io.FileInputStream.<init>(FileInputStream.java:106)
   at java.util.Scanner.<init>(Scanner.java:636)
   at FileSum.main(FileSum.java:17)
```

Rather than allowing the exception to be thrown and permitting this cryptic error message to be displayed, your program can check for the file's existence before it attempts to open the file. If the file does not exist, the program can display a more user-friendly error message and gracefully shut down.

After you create a `File` object representing the file that you want to open, you can use the `File` class's `exists` method to determine whether the file exists. The method returns `true` if the file exists, or `false` if the file does not exist. Code Listing 4-21 shows how to use the method. This is a modification of the *FileSum* program in Code Listing 4-20. This version of the program checks for the existence of the file *Numbers.txt* before it attempts to open it.

**Code Listing 4-21** `(FileSum2.java)`

```
import java.util.Scanner;
import java.io.*;

/**
   This version of the program confirms that the
   Numbers.txt file exists before opening it.
*/

public class FileSum2
{
    public static void main(String[] args) throws IOException
    {
       double sum = 0.0;  // Accumulator, initialized to 0
14
15       // Make sure the file exists.
16       File file = new File("Numbers.txt");
17       if (!file.exists())
18       {
19          System.out.println("The file Numbers.txt is not found.");
20          System.exit(0);
21       }
22
23       // Open the file for reading.
24       Scanner inputFile = new Scanner(file);
25
26       // Read all of the values from the file
27       // and calculate their total.
28       while (inputFile.hasNext())
29       {
30          // Read a value from the file.
31          double number = inputFile.nextDouble();
32
33          // Add the number to sum.
34          sum = sum + number;
35       }
36
37       // Close the file.
38       inputFile.close();
39
40       // Display the sum of the numbers.
41       System.out.println("The sum of the numbers in " +
42                          "Numbers.txt is " + sum);
43    }
44 }
```

**Program Output (Assuming Numbers.txt Does Not Exist)**

```
The file Numbers.txt is not found.
```

In line 16 the program creates a `File` object to represent the *Numbers.txt* file. In line 17, the `if` statement calls the `file.exists()` method. Notice the use of the `!` operator. If the method returns `false`, indicating that the file does not exist, the code in lines 19 and 20 executes. Line 19 displays an error message, and line 20 calls the `System.exit(0)` method to shut the program down.

The previous example shows you how to make sure that a file exists before trying to open it for input. But, when you are opening a file for output, sometimes you want to make sure the file does *not* exist. When you use a `PrintWriter` object to open a file, the file will be erased if it already exists. If you do not want to erase the existing file, you have to check for its existence before creating the `PrintWriter` object. Code Listing 4-22 shows you how to use the `File` class's `exists` method in this type of situation. This is a modification of the program you saw in Code Listing 4-17.

**Code Listing 4-22** (FileWriteDemo2.java)

```
import java.util.Scanner;      // Needed for Scanner class
import java.io.*;              // Needed for File and IOException

/**
   This program writes data to a file. It makes sure the
   specified file does not exist before opening it.
*/

public class FileWriteDemo2
{
   public static void main(String[] args) throws IOException
   {
      String filename;         // Filename
      String friendName;       // Friend's name
      int numFriends;          // Number of friends

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of friends.
      System.out.print("How many friends do you have? ");
      numFriends = keyboard.nextInt();

      // Consume the remaining newline character.
      keyboard.nextLine();

      // Get the filename.
      System.out.print("Enter the filename: ");
      filename = keyboard.nextLine();

      // Make sure the file does not exist.
      File file = new File(filename);
      if (file.exists())
      {
         System.out.println("The file " + filename +
                            " already exists.");
         System.exit(0);
      }

      // Open the file.
      PrintWriter outputFile = new PrintWriter(file);

      // Get data and write it to the file.
      for (int i = 1; i <= numFriends; i++)
      {
         // Get the name of a friend.
47          System.out.print("Enter the name of friend " +
48                           "number " + i + ": ");
49          friendName = keyboard.nextLine();
50
51          // Write the name to the file.
52          outputFile.println(friendName);
53        }
54
55        // Close the file.
56        outputFile.close();
57        System.out.println("Data written to the file.");
58     }
59  }
```

**Program Output with Example Input Shown in Bold**

```
How many friends do you have? 2 [Enter]
Enter the filename: MyFriends.txt [Enter]
The file MyFriends.txt already exists.
```

Line 32 creates a `File` object representing the file. The `if` statement in line 33 calls the `file.exists()` method. If the method returns `true`, then the file exists. In this case the code in lines 35 through 37 executes. This code displays an error message and shuts the program down. If the file does not exist, the rest of the program executes.

Notice that in line 41 we pass a reference to the `File` object to the `PrintWriter` constructor. In previous programs that created an instance `PrintWriter`, we passed a filename to the constructor. If you have a reference to a `File` object that represents the file you wish to open, as we do in this program, you have the option of passing it to the `PrintWriter` constructor.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.16 What is the difference between an input file and an output file?

4.17 What `import` statement will you need in a program that performs file operations?

4.18 What class do you use to write data to a file?

4.19 Write code that does the following: opens a file named *MyName.txt*, writes your first name to the file, and then closes the file.

4.20 What classes do you use to read data from a file?

4.21 Write code that does the following: opens a file named *MyName.txt*, reads the first line from the file and displays it, and then closes the file.

4.22 You are opening an existing file for output. How do you open the file without erasing it, and at the same time make sure that new data that is written to the file is appended to the end of the file's existing data?

4.23 What clause must you write in the header of a method that performs a file operation?

## 4.11 Generating Random Numbers with the Random Class

**CONCEPT: Random numbers are used in a variety of applications. Java provides the `Random` class that you can use to generate random numbers.**

Random numbers are useful for lots of different programming tasks. The following are just a few examples.

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice. Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.
- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.
- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

The Java API provides a class named `Random` that you can use to generate random numbers. The class is part of the `java.util` package, so any program that uses it will need an `import` statement such as:

```
import java.util.Random;
```

You create an object from the `Random` class with a statement such as this:

```
Random randomNumbers = new Random();
```

This statement does the following:

- It declares a variable named `randomNumbers`. The data type is the `Random` class.
- The expression `new Random()` creates an instance of the `Random` class.
- The equal sign assigns the address of the `Random` class to the `randomNumbers` variable.

After this statement executes, the `randomNumbers` variable will reference a `Random` object. Once you have created a `Random` object, you can call its `nextInt` method to get a random integer number. The following code shows an example:

```
// Declare an int variable.
int number;

// Create a Random object.
Random randomNumbers = new Random();

// Get a random integer and assign it to number.
number = randomNumbers.nextInt();
```

After this code executes, the `number` variable will contain a random integer. If you call the `nextInt` method with no arguments, as shown in this example, the returned integer is somewhere between −2,147,483,648 and + 2,147,483,647. Alternatively, you can pass an argument that specifies an upper limit to the generated number's range. In the following statement, the value assigned to `number` is somewhere between 0 and 99:

```
number = randomNumbers.nextInt(100);
```

You can add or subtract a value to shift the numeric range upward or downward. In the following statement, we call the `nextInt` method to get a random number in the range of 0 through 9, and then we add 1 to it. So, the number assigned to `number` will be somewhere in the range of 1 through 10:

```
number = randomNumbers.nextInt(10) + 1;
```

The following statement shows another example. It assigns a random integer to `number` between −50 and + 49:

```
number = randomNumbers.nextInt(100) - 50
```

The `Random` class has other methods for generating random numbers, and Table 4-1 summarizes several of them.

**Table 4-1** Some of the **Random** class's methods

| Method | Description |
|---|---|
| `nextDouble()` | Returns the next random number as a `double`. The number will be within the range of `0.0` through `1.0`. |
| `nextFloat()` | Returns the next random number as a `float`. The number will be within the range of `0.0` through `1.0`. |
| `nextInt()` | Returns the next random number as an `int`. The number will be within the range of an `int`, which is −2,147,483,648 to + 2,147,483,648. |
| `nextInt(int` *n*`)` | This method accepts an integer argument, *n*. It returns a random number as an `int`. The number will be within the range of 0 through *n*. |
| `nextLong()` | Returns the next random number as a `long`. The number will be within the range of a `long`, which is −9,223,372,036,854,775,808 to + 9,223,372,036,854,775,808. |

The program in Code Listing 4-23 demonstrates using the `Random` class.

**Code Listing 4-23** (`MathTutor.java`)

```
import java.util.Scanner;    // Needed for the Scanner class
import java.util.Random;     // Needed for the Random class

/**
   This program demonstrates the Random class.
*/

public class MathTutor
```

```
{
  public static void main(String[] args)
  {
     int number1;        // A number
     int number2;        // Another number
     int sum;            // The sum of the numbers
     int userAnswer;     // The user's answer

     // Create a Scanner object for keyboard input.
     Scanner keyboard = new Scanner(System.in);

     // Create a Random class object.
     Random randomNumbers = new Random();

     // Get two random numbers.
     number1 = randomNumbers.nextInt(100);
     number2 = randomNumbers.nextInt(100);

     // Display an addition problem.
     System.out.println("What is the answer to the " +
                        "following problem?");
     System.out.print(number1 + " + " +
                      number2 + " = ? ");

     // Calculate the answer.
     sum = number1 + number2;

     // Get the user's answer.
     userAnswer = keyboard.nextInt();

     // Display the user's results.
     if (userAnswer == sum)
        System.out.println("Correct!");
     else
     {
       System.out.println("Sorry, wrong answer. " +
                          "The correct answer is " +
                           sum);
     }
  }
}
```

**Program Output with Example Input Shown in Bold**

```
What is the answer to the following problem?
52 + 19 = ? 71 [Enter]
Correct!
```

**Program Output with Example Input Shown in Bold**

```
What is the answer to the following problem?
27 + 73 = ?  101 [Enter]
Sorry, wrong answer. The correct answer is 100
```

## In the Spotlight:

### Using Random Numbers

Dr. Kimura teaches an introductory statistics class, and has asked you to write a program that he can use in class to simulate the rolling of dice. The program should randomly generate two numbers in the range of 1 through 6 and display them. In your interview with Dr. Kimura, you learn that he would like to use the program to simulate several rolls of the dice, one after the other. Here is the pseudocode for the program:

```
While the user wants to roll the dice:
    Display a random number in the range of 1 through 6
    Display another random number in the range of 1 through 6
    Ask the user if he or she wants to roll the dice again
```

You will write a `while` loop that simulates one roll of the dice, and then asks the user whether another roll should be performed. As long as the user answers "y" for yes, the loop will repeat. Code Listing 4-24 shows the program.

**Code Listing 4-24** (`RollDice.java`)

```java
import java.util.Scanner;
import java.util.Random;

/**
   This program simulates the rolling of dice.
*/

public class RollDice
{
  public static void main(String[] args)
  {
    String again = "y";     // To control the loop
    int die1;               // To hold the value of die #1
    int die2;               // to hold the value of die #2

    // Create a Scanner object to read keyboard input.
    Scanner keyboard = new Scanner(System.in);

    // Create a Random object to generate random numbers.
    Random rand = new Random();

    // Simulate rolling the dice.
    while (again.equalsIgnoreCase("y"))
    {
       System.out.println("Rolling the dice . . . ");
       die1 = rand.nextInt(6) + 1;
       die2 = rand.nextInt(6) + 1;
       System.out.println("Their values are:");
       System.out.println(die1 + " " + die2);

       System.out.print("Roll them again (y = yes)? ");
       again = keyboard.nextLine();
    }
  }
}
```

**Program Output with Example Input Shown in Bold**

```
Rolling the dice . . .
Their values are:
4 3
Roll them again (y = yes)?   y [Enter]
Rolling the dice . . .
Their values are:
2 6
Roll them again (y = yes)?   y [Enter]
Rolling the dice . . .
Their values are:
1 5
Roll them again (y = yes)?   n [Enter]
```

## In the Spotlight:

## Using Random Numbers to Represent Other Values

Dr. Kimura was so happy with the dice rolling simulator that you wrote for him, he has asked you to write one more program. He would like a program that he can use to simulate ten coin tosses, one after the other. Each time the program simulates a coin toss, it should randomly display either "Heads" or "Tails".

You decide that you can simulate the tossing of a coin by randomly generating a number in the range of 0 through 1. You will write an `if` statement that displays "Tails" if the random number is 0, or "Heads" otherwise. Here is the pseudocode:

```
Repeat 10 times:
   If a random number in the range of 0 through 1 equals 0, then:
      Display "Tails"
   Else:
      Display "Heads"
```

Because the program should simulate 10 tosses of a coin, you decide to use a `for` loop. The program is shown in Code Listing 4-25.

**Code Listing 4-25** (`CoinToss.java`)

```java
import java.util.Random;

/**
   This program simulates 10 tosses of a coin.
*/

public class CoinToss
{
   public static void main(String[] args)
   {
      // Create a Random object to generate random numbers.
      Random rand = new Random();

      // Simulate the coin tosses.
      for (int count = 0; count < 10; count++)
      {
         if (rand.nextInt(2) == 0)
            System.out.println("Tails");
         else
           System.out.println("Heads");
      }
  }
}
```

**Program Output**

```
Tails
Tails
Heads
Tails
Heads
Heads
Heads
Tails
Heads
Tails
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

4.24 Assume `x` is an `int` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextInt();
```

4.25 Assume `x` is an `int` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextInt(100);
```

4.26 Assume `x` is an `int` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextInt(9) + 1;
```

4.27 Assume `x` is a `double` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextDouble();
```

## 4.12 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics.

- **Using the increment or decrement operator in the wrong mode.** When the increment or decrement operator is placed in front of (to the left of) its operand, it is used in prefix mode. When either of these operators is placed behind (to the right of) its operand, it is used in postfix mode.
- **Forgetting to enclose the `boolean` expression in a `while` loop or a `do-while` loop inside parentheses.**
- **Placing a semicolon at the end of a `while` or `for` loop's header.** When you write a semicolon at the end of a `while` or `for` loop's header, Java assumes that the conditionally executed statement is a `null` or empty statement. This usually results in an infinite loop.
- **Forgetting to write the semicolon at the end of the `do-while` loop.** The `do-while` loop must be terminated with a semicolon.
- **Forgetting to enclose multiple statements in the body of a loop in braces.** Normally a loop conditionally executes only one statement. To conditionally execute more than one statement, you must place the statements in braces.
- **Using commas instead of semicolons to separate the initialization, test, and update expressions in a `for` loop.**
- **Forgetting to write code in the body of a `while` or `do-while` loop that modifies the loop control variable.** If a `while` or `do-while` loop's `boolean` expression never becomes `false`, the loop will repeat indefinitely. You must have code in the body of the loop that modifies the loop control variable so that the `boolean` expression will at some point become `false`.
- **Using a sentinel value that can also be a valid data value.** Remember, a sentinel is a special value that cannot be mistaken as a member of a list of data items and signals that there are no more data items from the list to be processed. If you choose as a sentinel a value that might also appear in the list, the loop will prematurely terminate if it encounters the value in the list.
- **Forgetting to initialize an accumulator to zero.** In order for an accumulator to keep a correct running total, it must be initialized to zero before any values are added to it.

## Review Questions and Exercises

### Multiple Choice and True/False

1. What will the `println` statement in the following program segment display?

   ```
   int x = 5;
   System.out.println(x++);
   ```

   a. 5
   b. 6
   c. 0
   d. None of these

2. What will the `println` statement in the following program segment display?

   ```
   int x = 5;
   System.out.println(++x);
   ```

   a. 5
   b. 6
   c. 0
   d. None of these

3. In the expression `number++`, the `++` operator is in what mode?
   a. prefix
   b. pretest
   c. postfix
   d. posttest

4. What is each repetition of a loop known as?
   a. cycle
   b. revolution
   c. orbit
   d. iteration

5. This is a variable that controls the number of iterations performed by a loop.
   a. loop control variable
   b. accumulator
   c. iteration register variable
   d. repetition meter

6. The `while` loop is this type of loop.
   a. pretest
   b. posttest
   c. prefix
   d. postfix

7. The `do-while` loop is this type of loop.
   a. pretest
   b. posttest
   c. prefix
   d. postfix

8. The `for` loop is this type of loop.
   a. pretest
   b. posttest
   c. prefix
   d. postfix
9. This type of loop has no way of ending and repeats until the program is interrupted.
   a. indeterminate
   b. interminable
   c. infinite
   d. timeless
10. This type of loop always executes at least once.
    a. `while`
    b. `do-while`
    c. `for`
    d. any of these
11. This expression is executed by the `for` loop only once, regardless of the number of iterations.
    a. initialization expression
    b. test expression
    c. update expression
    d. pre-increment expression
12. This is a variable that keeps a running total.
    a. sentinel
    b. sum
    c. total
    d. accumulator
13. This is a special value that signals when there are no more items from a list of items to be processed. This value cannot be mistaken as an item from the list.
    a. sentinel
    b. flag
    c. signal
    d. accumulator
14. To open a file for writing, you use the following class.
    a. `PrintWriter`
    b. `FileOpen`
    c. `OutputFile`
    d. `FileReader`
15. To open a file for reading, you use the following classes.
    a. `File` and `Writer`
    b. `File` and `Output`
    c. `File` and `Input`
    d. `File` and `Scanner`

16. When a program is finished using a file, it should do this.
    a. erase the file
    b. close the file
    c. throw an exception
    d. reset the read position
17. This class allows you to use the `print` and `println` methods to write data to a file.
    a. `File`
    b. `FileReader`
    c. `OutputFile`
    d. `PrintWriter`
18. This class allows you to read a line from a file.
    a. `FileWriter`
    b. `Scanner`
    c. `InputFile`
    d. `FileReader`
19. True or False: The `while` loop is a pretest loop.
20. True or False: The `do-while` loop is a pretest loop.
21. True or False: The `for` loop is a posttest loop.
22. True or False: It is not necessary to initialize accumulator variables.
23. True or False: One limitation of the `for` loop is that only one variable may be initialized in the initialization expression.
24. True or False: A variable may be defined in the initialization expression of the `for` loop.
25. True or False: In a nested loop, the inner loop goes through all of its iterations for every iteration of the outer loop.
26. True or False: To calculate the total number of iterations of a nested loop, add the number of iterations of all the loops.

### Find the Error

Find the errors in the following code:

1.
```
// This code contains ERRORS!
// It adds two numbers entered by the user.
int num1, num2;
String input;
char again;

Scanner keyboard = new Scanner(System.in);
while (again == 'y' || again == 'Y')
   System.out.print("Enter a number: ");
   num1 = keyboard.nextInt();
   System.out.print("Enter another number: ";
   num2 = keyboard.nextInt();
   System.out.println("Their sum is "+ (num1 + num2));
   System.out.println("Do you want to do this again? ");
   keyboard.nextLine(); // Consume remaining newline
   input = keyboard.nextLine();
   again = input.charAt(0);
```

2.
```
// This code contains ERRORS!
int count = 1, total;
while (count <= 100)
  total += count;
System.out.print("The sum of the numbers 1 - 100 is ");
System.out.println(total);
```

3.
```
// This code contains ERRORS!
int choice, num1, num2;
Scanner keyboard = new Scanner(System.in);
do
{
   System.out.print("Enter a number: ");
   num1 = keyboard.nextInt();
   System.out.print("Enter another number: ");
   num2 = keyboard.nextInt();
   System.out.println("Their sum is " + (num1 + num2));
   System.out.println("Do you want to do this again? ");
   System.out.print("1 = yes, 0 = no ");
   choice = keyboard.nextInt();
} while (choice = 1)
```

4.
```
// This code contains ERRORS!
// Print the numbers 1 through 10.
for (int count = 1, count <= 10, count++;)
{
       System.out.println(count);
       count++;
}
```

## Algorithm Workbench

1. Write a `while` loop that lets the user enter a number. The number should be multiplied by 10, and the result stored in the variable `product`. The loop should iterate as long as `product` contains a value less than 100.
2. Write a `do-while` loop that asks the user to enter two numbers. The numbers should be added and the sum displayed. The loop should ask the user whether he or she wishes to perform the operation again. If so, the loop should repeat; otherwise it should terminate.
3. Write a `for` loop that displays the following set of numbers:

   `0, 10, 20, 30, 40, 50 . . . 1000`

4. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.
5. Write a `for` loop that calculates the total of the following series of numbers:

   $$\frac{1}{30} + \frac{2}{29} + \frac{3}{28} + \cdots \frac{30}{1}$$

6. Write a nested loop that displays 10 rows of `'#'` characters. There should be 15 `'#'` characters in each row.
7. Convert the `while` loop in the following code to a `do-while` loop:

```
Scanner keyboard = new Scanner(System.in);
int x = 1;
while (x > 0)
{
   System.out.print("Enter a number: ");
   x = keyboard.nextInt();
}
```

8. Convert the `do-while` loop in the following code to a `while` loop:

```
Scanner keyboard = new Scanner(System.in);
String input;
char sure;
do
{
   System.out.print("Are you sure you want to quit? ");
   input = keyboard.next();
   sure = input.charAt(0);
} while (sure != 'Y' && sure != 'N');
```

9. Convert the following `while` loop to a `for` loop:

```
int count = 0;
while (count < 50)
{
  System.out.println("count is " + count);
  count++;
}
```

10. Convert the following `for` loop to a `while` loop:

```
for (int x = 50; x > 0; x--)
{
   System.out.println(x + " seconds to go.");
}
```

11. Write an input validation loop that asks the user to enter a number in the range of 1 through 4.
12. Write an input validation loop that asks the user to enter the word “yes” or “no”.

13. Write nested loops to draw this pattern:

```
*******
******
*****
****
***
**
*
```

14. Write nested loops to draw this pattern:

```
##
# #
#  #
#   #
#    #
#     #
```

15. Complete the following program so it displays a random integer in the range of 1 through 10.

```
// Write the necessary import statement(s) here.
public class ReviewQuestion15
{
   public static void main(String[] args)
   {
      // Write the necessary code here.
   }
}
```

16. Complete the following program so it performs the following actions 10 times:
    - Generates a random number that is either 0 or 1.
    - Displays either the word "Yes" or the word "No" depending on the random number that was generated.

```
// Write the necessary import statement(s) here.
public class ReviewQuestion16
{
   public static void main(String[] args)
   {
      // Write the necessary code here.
   }
}
```

17. Write code that does the following: opens a file named *NumberList.txt*, uses a loop to write the numbers 1 through 100 to the file, and then closes the file.
18. Write code that does the following: opens the *NumberList.txt* file that was created by the code in Question 17, reads all of the numbers from the file and displays them, and then closes the file.
19. Modify the code you wrote in Question 18 so it adds all of the numbers read from the file and displays their total.
20. Write code that opens a file named *NumberList.txt* for writing, but does not erase the file's contents if it already exists.

### Short Answer

1. Briefly describe the difference between the prefix and postfix modes used by the increment and decrement operators.
2. Why should you indent the statements in the body of a loop?
3. Describe the difference between pretest loops and posttest loops.
4. Why are the statements in the body of a loop called conditionally executed statements?
5. Describe the difference between the `while` loop and the `do-while` loop.
6. Which loop should you use in situations where you want the loop to repeat until the `boolean` expression is `false`, and the loop should not execute if the test expression is `false` to begin with?
7. Which loop should you use in situations where you want the loop to repeat until the `boolean` expression is `false`, but the loop should execute at least once?
8. Which loop should you use when you know the number of required iterations?
9. Why is it critical that accumulator variables are properly initialized?
10. What is an infinite loop? Write the code for an infinite loop.
11. Describe a programming problem that would require the use of an accumulator.
12. What does it mean to let the user control a loop?
13. What is the advantage of using a sentinel?
14. Why must the value chosen for use as a sentinel be carefully selected?
15. Describe a programming problem requiring the use of nested loops.
16. How does a file buffer increase a program's performance?
17. Why should a program close a file when it's finished using it?
18. What is a file's read position? Where is the read position when a file is first opened for reading?
19. When writing data to a file, what is the difference between the `print` and the `println` methods?
20. What does the `Scanner` class's `hasNext` method return when the end of the file has been reached?
21. What is a potential error that can occur when a file is opened for reading?
22. What does it mean to append data to a file?
23. How do you open a file so that new data will be written to the end of the file's existing data?

## Programming Challenges

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Sum of Numbers

Write a program that asks the user for a positive nonzero integer value. The program should use a loop to get the sum of all the integers from 1 up to the number entered. For example, if the user enters 50, the loop will find the sum of 1, 2, 3, 4, . . . 50.

### 2. Distance Traveled

The distance a vehicle travels can be calculated as follows:

$$\text{Distance} = \text{Speed} * \text{Time}$$

For example, if a train travels 40 miles-per-hour for three hours, the distance traveled is 120 miles. Write a program that asks for the speed of a vehicle (in miles-per-hour) and the number of hours it has traveled. It should use a loop to display the distance a vehicle has traveled for each hour of a time period specified by the user. For example, if a vehicle is traveling at 40 mph for a three-hour time period, it should display a report similar to the one that follows:

```
 Hour        Distance Traveled
-------------------------------
 1                  40
 2                  80
 3                 120
```

*Input Validation: Do not accept a negative number for speed and do not accept any value less than 1 for time traveled.*

### 3. Distance File

Modify the program you wrote for Programming Challenge 2 (Distance Traveled) so it writes the report to a file instead of the screen. Open the file in Notepad or another text editor to confirm the output.

### 4. Pennies for Pay

VideoNote
The Pennies for Pay Problem

Write a program that calculates the amount a person would earn over a period of time if his or her salary is one penny the first day, two pennies the second day, and continues to double each day. The program should display a table showing the salary for each day, and then show the total pay at the end of the period. The output should be displayed in a dollar amount, not the number of pennies.

*Input Validation: Do not accept a number less than 1 for the number of days worked.*

### 5. Letter Counter

Write a program that asks the user to enter a string, and then asks the user to enter a character. The program should count and display the number of times that the specified character appears in the string.

### 6. File Letter Counter

Write a program that asks the user to enter the name of a file, and then asks the user to enter a character. The program should count and display the number of times that the specified character appears in the file. Use Notepad or another text editor to create a simple file that can be used to test the program.

### 7. Hotel Occupancy

A hotel's occupancy rate is calculated as follows:

$$\text{Occupancy rate} = \text{Number of rooms occupied} \div \text{Total number of rooms}$$

Write a program that calculates the occupancy rate for each floor of a hotel. The program should start by asking for the number of floors in the hotel. A loop should then iterate once for each floor. During each iteration, the loop should ask the user for the number of rooms on the floor and the number of them that are occupied. After all the iterations, the program should display the number of rooms the hotel has, the number of them that are occupied, the number that are vacant, and the occupancy rate for the hotel.

*Input Validation: Do not accept a value less than 1 for the number of floors. Do not accept a number less than 10 for the number of rooms on a floor.*

### 8. Average Rainfall

Write a program that uses nested loops to collect data and calculate the average rainfall over a period of years. First the program should ask for the number of years. The outer loop will iterate once for each year. The inner loop will iterate 12 times, once for each month. Each iteration of the inner loop will ask the user for the inches of rainfall for that month. After all iterations, the program should display the number of months, the total inches of rainfall, and the average rainfall per month for the entire period.

*Input Validation: Do not accept a number less than 1 for the number of years. Do not accept negative numbers for the monthly rainfall.*

### 9. Population

Write a program that will predict the size of a population of organisms. The program should ask for the starting number of organisms, their average daily population increase (as a percentage), and the number of days they will multiply. For example, a population might begin with two organisms, have an average daily increase of 50 percent, and will be allowed to multiply for seven days. The program should use a loop to display the size of the population for each day.

*Input Validation: Do not accept a number less than 2 for the starting size of the population. Do not accept a negative number for average daily population increase. Do not accept a number less than 1 for the number of days they will multiply.*

### 10. Largest and Smallest

Write a program with a loop that lets the user enter a series of integers. The user should enter −99 to signal the end of the series. After all the numbers have been entered, the program should display the largest and smallest numbers entered.

### 11. Celsius to Fahrenheit Table

Write a program that displays a table of the Celsius temperatures 0 through 20 and their Fahrenheit equivalents. The formula for converting a temperature from Celsius to Fahrenheit is

$$F = \frac{9}{5}C + 32$$

where $F$ is the Fahrenheit temperature and $C$ is the Celsius temperature. Your program must use a loop to display the table.

### 12. Bar Chart

Write a program that asks the user to enter today's sales for five stores. The program should display a bar chart comparing each store's sales. Create each bar in the bar chart by displaying a row of asterisks. Each asterisk should represent $100 of sales. Here is an example of the program's output:

```
Enter today's sales for store 1: 1000 [Enter]
Enter today's sales for store 2: 1200 [Enter]
Enter today's sales for store 3: 1800 [Enter]
Enter today's sales for store 4: 800 [Enter]
Enter today's sales for store 5: 1900 [Enter]

SALES BAR CHART
Store 1: **********
Store 2: ************
Store 3: ******************
Store 4: ********
Store 5: *******************
```

### 13. File Head Display

Write a program that asks the user for the name of a file. The program should display only the first five lines of the file's contents. If the file contains fewer than five lines, it should display the file's entire contents.

### 14. Line Numbers

Write a program that asks the user for the name of a file. The program should display the contents of the file with each line preceded with a line number followed by a colon. The line numbering should start at 1.

### 15. Uppercase File Converter

Write a program that asks the user for the names of two files. The first file should be opened for reading and the second file should be opened for writing. The program should read the contents of the first file, change all characters to uppercase, and store the results in the second file. The second file will be a copy of the first file, except that all the characters will be uppercase. Use Notepad or another text editor to create a simple file that can be used to test the program.

### 16. Budget Analysis

Write a program that asks the user to enter the amount that he or she has budgeted for a month. A loop should then prompt the user to enter each of his or her expenses for the month, and keep a running total. When the loop finishes, the program should display the amount that the user is over or under budget.

### 17. Random Number Guessing Game

Write a program that generates a random number and asks the user to guess what the number is. If the user's guess is higher than the random number, the program should display "Too high, try again." If the user's guess is lower than the random number, the program should display "Too low, try again." The program should use a loop that repeats until the user correctly guesses the random number.

### 18. Random Number Guessing Game Enhancement

Enhance the program that you wrote for Programming Challenge 17 so it keeps a count of the number of guesses that the user makes. When the user correctly guesses the random number, the program should display the number of guesses.

### 19. ESP Game

Write a program that tests your ESP (extrasensory perception). The program should randomly select the name of a color from the following list of words:

*Red, Green, Blue, Orange, Yellow*

To select a word, the program can generate a random number. For example, if the number is 0, the selected word is *Red*, if the number is 1, the selected word is *Green*, and so forth.

Next, the program should ask the user to enter the color that the computer has selected. After the user has entered his or her guess, the program should display the name of the randomly selected color. The program should repeat this 10 times and then display the number of times the user correctly guessed the selected color.

### 20. Square Display

Write a program that asks the user for a positive integer no greater than 15. The program should then display a square on the screen using the character 'X'. The number entered by the user will be the length of each side of the square. For example, if the user enters 5, the program should display the following:

```
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

If the user enters 8, the program should display the following:

```
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
```

### 21. Dice Game

Write a program that plays a simple dice game between the computer and the user. When the program runs, a loop should repeat 10 times. Each iteration of the loop should do the following:

- Generate a random integer in the range of 1 through 6. This is the value of the computer's die.
- Generate another random integer in the range of 1 through 6. This is the value of the user's die.
- The die with the highest value wins. (In case of a tie, there is no winner for that particular roll of the dice.)

As the loop iterates, the program should keep count of the number of times the computer wins, and the number of times that the user wins. After the loop performs all of its iterations, the program should display who was the grand winner, the computer or the user.

### 22. Slot Machine Simulation

A slot machine is a gambling device that the user inserts money into and then pulls a lever (or presses a button). The slot machine then displays a set of random images. If two or more of the images match, the user wins an amount of money that the slot machine dispenses back to the user.

Create a program that simulates a slot machine. When the program runs, it should do the following:

- Asks the user to enter the amount of money he or she wants to enter into the slot machine.
- Instead of displaying images, the program will randomly select a word from the following list:
  *Cherries, Oranges, Plums, Bells, Melons, Bars*
- To select a word, the program can generate a random number in the range of 0 through 5. If the number is 0, the selected word is *Cherries*; if the number is 1, the selected word is *Oranges*; and so forth. The program should randomly select a word from this list three times and display all three of the words.
- If none of the randomly selected words match, the program will inform the user that he or she has won $0. If two of the words match, the program will inform the user that he or she has won two times the amount entered. If three of the words match, the program will inform the user that he or she has won three times the amount entered.
- The program will ask whether the user wants to play again. If so, these steps are repeated. If not, the program displays the total amount of money entered into the slot machine and the total amount won. Decision Structures

### 23. Personal Web Page Generator

Write a program that asks the user for his or her name, and then asks the user to enter a sentence that describes him or herself. Here is an example of the program's screen:

```
Enter your name: Julie Taylor [Enter]

Describe yourself: I am a computer science major, a member of the Jazz club,
and I hope to work as a mobile app developer after I graduate. [Enter]
```

Once the user has entered the requested input, the program should create an HTML file, containing the input, for a simple Web page. Here is an example of the HTML content, using the sample input previously shown:

```
<html>
<head>
</head>
<body>
   <center>
      <h1>Julie Taylor</h1>
   </center>
   <hr />
   I am a computer science major, a member of the Jazz club,
   and I hope to work as a mobile app developer after I graduate.
   <hr />
</body>
</html>
```

CHAPTER

# 5 Methods

**TOPICS**

5.1 Introduction to Methods
5.2 Passing Arguments to a Method
5.3 More about Local Variables
5.4 Returning a Value from a Method
5.5 Problem Solving with Methods
5.6 Common Errors to Avoid

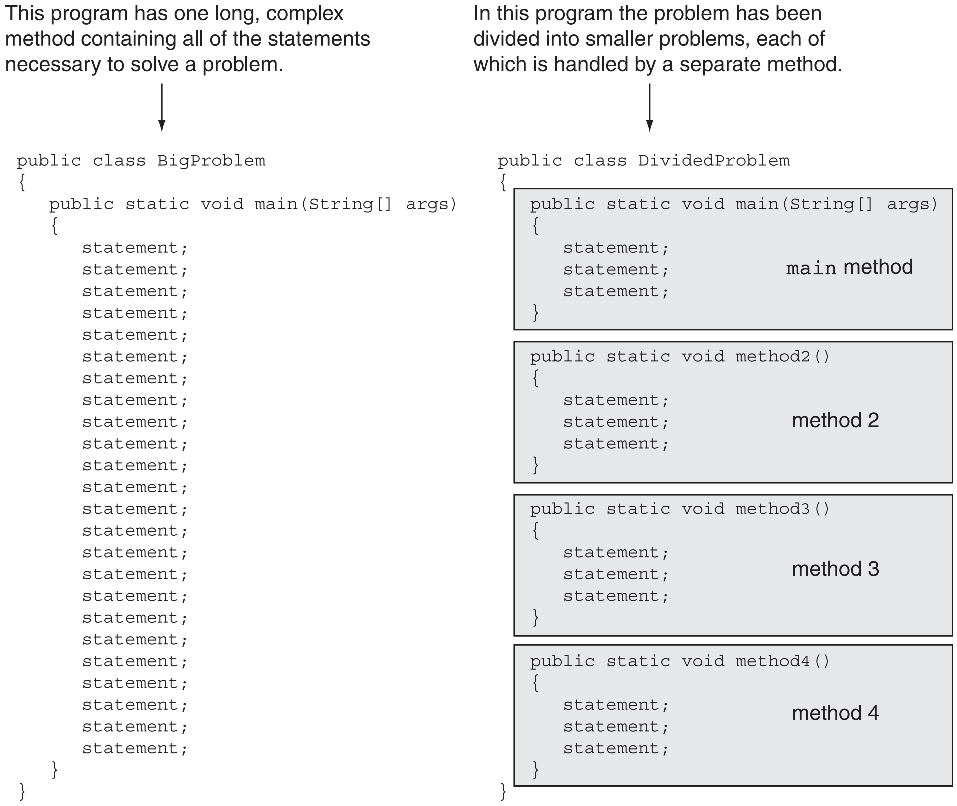
## 5.1 Introduction to Methods

**CONCEPT: Methods can be used to break a complex program into small, manageable pieces. A `void` method simply executes a group of statements and then terminates. A value-returning method returns a value to the statement that called it.**

In a general sense, a method is a collection of statements that performs a specific task. So far you have experienced methods in two ways: (1) You have created a method named `main` in every program you've written, and (2) you have executed predefined methods from the Java API, such as `System.out.println`, `Integer.parseInt`, and `Math.pow`. In this chapter, you will learn how to create your own methods, other than `main`, that can be executed just as you execute the API methods.

Methods are commonly used to break a problem into small, manageable pieces. Instead of writing one long method that contains all of the statements necessary to solve a problem, several small methods that each solve a specific part of the problem can be written. These small methods can then be executed in the desired order to solve the problem. This approach is sometimes called *divide and conquer* because a large problem is divided into several smaller problems that are easily solved. Figure 5-1 illustrates this idea by comparing two programs: one that uses a long, complex method containing all of the statements necessary to solve a problem, and another that divides a problem into smaller problems, each of which is handled by a separate method.

**Figure 5-1** Using methods to divide and conquer a problem



This program has one long, complex method containing all of the statements necessary to solve a problem.

```
public class BigProblem
{
   public static void main(String[] args)
   {
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
   }
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate method.

```
public class DividedProblem
{
   public static void main(String[] args)
   {
      statement;
      statement;
      statement;
   }

   public static void method2()
   {
      statement;
      statement;
      statement;
   }

   public static void method3()
   {
      statement;
      statement;
      statement;
   }

   public static void method4()
   {
      statement;
      statement;
      statement;
   }
}
```

Another reason to write methods is that they simplify programs. If a specific task is performed in several places in a program, a method can be written once to perform that task, and then be executed any time it is needed. This benefit of using methods is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

First, we will look at the general ways in which methods operate. At the end of the chapter we will discuss in greater detail how methods can be used in problem solving.

## `void` Methods and Value-Returning Methods

In this chapter, you will learn about two general categories of methods: `void` methods and value-returning methods. A *`void` method* is one that simply performs a task and then terminates. `System.out.println` is an example of a `void` method. For example, look at the following code:

```
int number = 7;
System.out.println(number);
number = 0;
```

The statement in line 1 declares the `number` variable and initializes it with the value 7. The statement in line 2 calls the `System.out.println` method, passing `number` as an argument. The method does its job, which is to display a value on the screen, and then terminates. The code then resumes at line 3.

A *value-returning method* not only performs a task but also sends a value back to the code that called it. The `Random` class's `nextInt` method is an example of a value-returning method. For example, look at the following code:

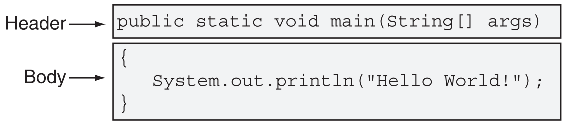
```
1    int number;
2    Random rand = new Random();
3    number = rand.nextInt();
```

The statement in line 1 declares the `number` variable. Line 2 creates a `Random` object and assigns its address to a variable named `rand`. Line 3 is an assignment statement, which assigns a value to the `number` variable. Notice that on the right side of the `=` operator is a call to the `rand.nextInt` method. The method executes, and then returns a value. The value that is returned from the method is assigned to the `number` variable.

## Defining a `void` Method

To create a method you must write its *definition*, which consists of two general parts: a header and a body. You learned about both of these in Chapter 2, but let's briefly review. The *method header*, which appears at the beginning of a method definition, lists several important things about the method, including the method's name. The *method body* is a collection of statements that are performed when the method is executed. These statements are enclosed inside a set of curly braces. Figure 5-2 points out the header and body of a `main` method.

**Figure 5-2** The header and body of a `main` method



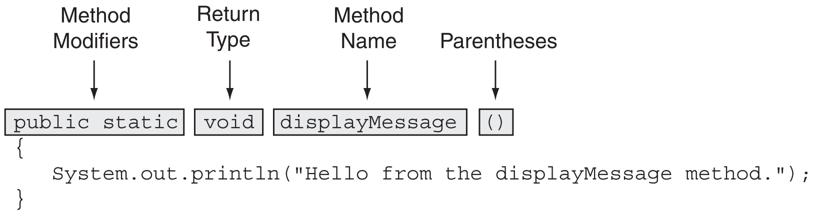
```
public static void main(String[] args)
{
   System.out.println("Hello World!");
}
```

As you already know, every complete Java program must have a `main` method. Java programs can have other methods as well. Here is an example of a simple method that displays a message on the screen:

```
public static void displayMessage()
{
    System.out.println("Hello from the displayMessage method.");
}
```

This method has a header and a body. Figure 5-3 shows the different parts of the method header.

**Figure 5-3** Parts of the method header



```
public static void displayMessage ()
{
   System.out.println("Hello from the displayMessage method.");
}
```

Let's take a closer look at the parts identified in the figure as follows:

- **Method modifiers**—The key words `public` and `static` are modifiers. You don't need to be too concerned with these modifiers now, but if your curiosity is getting the best of you, here's a brief explanation: The word `public` means that the method is publicly available to code outside the class. The word `static` means that the method belongs to the class, not a specific object. You will learn more about these modifiers in later chapters. For this chapter, every method that we write will begin with `public static`.
- **Return type**—Recall our previous discussion of `void` and value-returning methods. When the key word `void` appears here, it means that the method is a `void` method, and does not return a value. As you will see later, a value-returning method lists a data type here.
- **Method name**—You should give each method a descriptive name. In general, the same rules that apply to variable names also apply to method names. This method is named `displayMessage`, so we can easily guess what the method does: It displays a message.
- **Parentheses**—In the header, the method name is always followed by a set of parentheses. As you will learn later in this chapter, methods can be capable of receiving arguments. When this is the case, a list of one or more variable declarations will appear inside the parentheses. The method in this example does not receive any arguments, so the parentheses are empty.

**NOTE:** The method header is never terminated with a semicolon.

## Calling a Method

A method executes when it is called. The `main` method is automatically called when a program starts, but other methods are executed by method call statements. When a method is called, the JVM branches to that method and executes the statements in its body. Here is an example of a method call statement that calls the `displayMessage` method we previously examined:

```
displayMessage();
```

The statement is simply the name of the method followed by a set of parentheses. Because it is a complete statement, it is terminated with a semicolon.

**TIP:** Notice that the method modifiers and the `void` return type are not written in the method call statement. They are written only in the method header.

The program in Code Listing 5-1 demonstrates.

**Code Listing 5-1** `(SimpleMethod.java)`

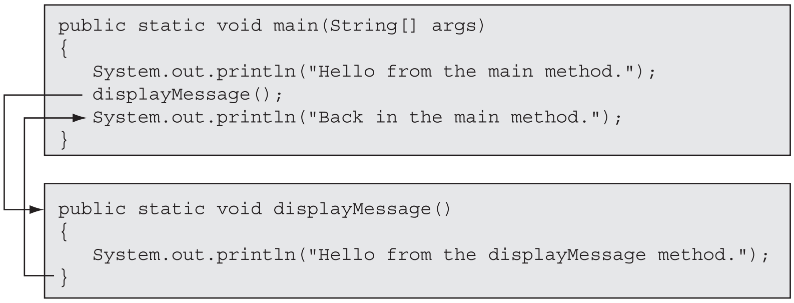
```
 1  /**
 2     This program defines and calls a simple method.
 3  */
 4
 5  public class SimpleMethod
 6  {
 7     public static void main(String[] args)
 8     {
 9        System.out.println("Hello from the main method.");
10        displayMessage();
11        System.out.println("Back in the main method.");
12     }
13
14     /**
15        The displayMessage method displays a greeting.
16     */
17
18     public static void displayMessage()
19     {
20        System.out.println("Hello from the displayMessage method.");
21     }
22  }
```

**Program Output**

```
Hello from the main method.
Hello from the displayMessage method.
Back in the main method.
```

Notice how the program flows. It starts, of course, in the `main` method. When the call to the `displayMessage` method in line 10 is encountered, the JVM branches to that method and performs the statement in its body (at line 20). Once the `displayMessage` method has finished executing, the JVM branches back to the `main` method and resumes at line 11 with the statement that follows the method call. This is illustrated in Figure 5-4.

**Figure 5-4** Branching in the `SimpleMethod.java` program



```
public static void main(String[] args)
{
   System.out.println("Hello from the main method.");
   displayMessage();
   System.out.println("Back in the main method.");
}
```

```
public static void displayMessage()
{
   System.out.println("Hello from the displayMessage method.");
}
```

Method call statements may be used in control structures like loops, `if` statements, and `switch` statements. The program in Code Listing 5-2 places the `displayMessage` method call inside a loop.

**Code Listing 5-2** `(LoopCall.java)`

```
/**
  This program defines and calls a simple method.
*/

public class LoopCall
{
   public static void main(String[] args)
   {
      System.out.println("Hello from the main method.");
      for (int i = 0; i < 5; i++)
         displayMessage();
      System.out.println("Back in the main method.");
   }

   /**
      The displayMessage method displays a greeting.
   */

   public static void displayMessage()
   {
      System.out.println("Hello from the displayMessage method.");
   }
}
```

**Program Output**

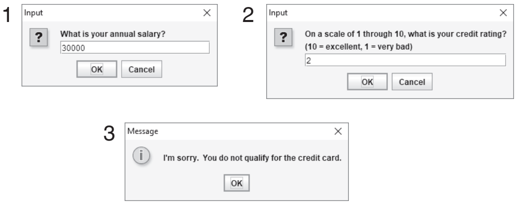
```
Hello from the main method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Back in the main method.
```

The program in Code Listing 5-3 shows another example. It asks the user to enter his or her annual salary and credit rating. The program then determines whether the user qualifies for a credit card. One of two `void` methods, `qualify` or `noQualify`, is called to display a message. Figures 5-5 and 5-6 show example interactions with the program.

**Code Listing 5-3** (`CreditCard.java`)

```
import javax.swing.JOptionPane;

/**
  This program uses two void methods.
*/

public class CreditCard
{
   public static void main(String[] args)
   {
      double salary;     // Annual salary
      int creditRating;  // Credit rating
      String input;      // To hold the user's input

      // Get the user's annual salary.
      input = JOptionPane.showInputDialog("What is " +
                                   "your annual salary?");
      salary = Double.parseDouble(input);

      // Get the user's credit rating (1 through 10).
      input = JOptionPane.showInputDialog("On a scale of " +
            "1 through 10, what is your credit rating?\n" +
            "(10 = excellent, 1 = very bad)");
      creditRating = Integer.parseInt(input);

      // Determine whether the user qualifies.
      if (salary >= 20000 && creditRating >= 7)
         qualify();
      else
         noQualify();

      System.exit(0);
   }

   /**
     The qualify method informs the user that he
     or she qualifies for the credit card.
   */

   public static void qualify()
   {
      JOptionPane.showMessageDialog(null, "Congratulations! " +
                                   "You qualify for the credit card!");
   }

   /**
      The noQualify method informs the user that he
      or she does not qualify for the credit card.
   */

   public static void noQualify()
   {
      JOptionPane.showMessageDialog(null, "I'm sorry. You " +
                                "do not qualify for the credit card.");
   }
}
```

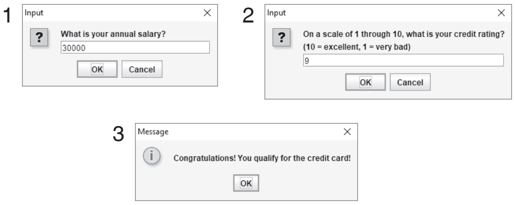
**Figure 5-5** Interaction with the `CreditCard.java` program (Oracle Corporate Counsel)


1 What is your annual salary? 30000

2 On a scale of 1 through 10, what is your credit rating? (10 = excellent, 1 = very bad) 2

3 I'm sorry. You do not qualify for the credit card.

**Figure 5-6** Interaction with the `CreditCard.java` program (Oracle Corporate Counsel)


1 What is your annual salary? 30000

2 On a scale of 1 through 10, what is your credit rating? (10 = excellent, 1 = very bad) 9

3 Congratulations! You qualify for the credit card!

## Layered Method Calls

Methods can also be called in layered fashion. In other words, method A can call method B, which can then call method C. When method C finishes, the JVM returns to method B. When method B finishes, the JVM returns to method A. The program in Code Listing 5-4 demonstrates this with three methods: `main`, `deep`, and `deeper`. The `main` method calls the `deep` method, which then calls the `deeper` method.

**Code Listing 5-4** `(DeepAndDeeper.java)`

```java
/**
  This program demonstrates layered method calls.
*/

public class DeepAndDeeper
{
   public static void main(String[] args)
   {
      System.out.println("I am starting in main.");
      deep();
      System.out.println("Now I am back in main.");
   }

   /**
      The deep method displays a message and then calls
      the deeper method.
   */

   public static void deep()
   {
      System.out.println("I am now in deep.");
      deeper();
      System.out.println("Now I am back in deep.");
   }

   /**
      The deeper method simply displays a message.
   */

   public static void deeper()
   {
      System.out.println("I am now in deeper.");
   }
}
```

**Program Output**

```
I am starting in main.
I am now in deep.
I am now in deeper.
Now I am back in deep.
Now I am back in main.
```

## Using Documentation Comments with Methods

You should always document a method by writing comments that appear just before the method's definition. The comments should provide a brief explanation of the method's purpose. Notice that the programs we've looked at in this chapter use documentation comments.

Recall from Chapter 2 that documentation comments begin with `/**` and end with `*/`. These types of comments can be read and processed by a program named `javadoc`, which produces attractive HTML documentation. As we progress through this chapter, you will learn more about documentation comments and how they can be used with methods.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

5.1 What is the difference between a `void` method and a value-returning method?

5.2 Is the following line of code a method header or a method call?

```
calcTotal();
```

5.3 Is the following line of code a method header or a method call?

```
public static void calcTotal()
```

5.4 What message will the following program display if the user enters 5? What if the user enters 10? What if the user enters 100?

```
import javax.swing.JOptionPane;
public class Checkpoint
{
   public static void main(String[] args)
   {
      String input;
      int number;

      input = JOptionPane.showInputDialog("Enter a number.");
      number = Integer.parseInt(input);

      if (number < 10)
         method1();
      else
         method2();

      System.exit(0);
   }

   public static void method1()
   {
      JOptionPane.showMessageDialog(null, "Able was I.");
   }
   public static void method2()
   {
      JOptionPane.showMessageDialog(null, "I saw Elba.");
   }
}
```

5.5 Write a `void` method that displays your full name. The method should be named `myName`.

## 5.2 Passing Arguments to a Method

**CONCEPT: A method may be written so it accepts arguments. Data can then be passed into the method when it is called.**

VideoNote
Passing Arguments to a Method

Values that are sent into a method are called *arguments*. You're already familiar with how to use arguments in a method call. For example, look at the following statement:

```
System.out.println("Hello");
```

This statement calls the `System.out.println` method and passes `"Hello"` as an argument. Here is another example:

```
number = Integer.parseInt(str);
```

This statement calls the `Integer.parseInt` method and passes the contents of the `str` variable as an argument. By using parameter variables, you can design your own methods that accept data this way. A *parameter variable*, sometimes simply referred to as a *parameter*, is a special variable that holds a value being passed into a method. Here is the definition of a method that uses a parameter:

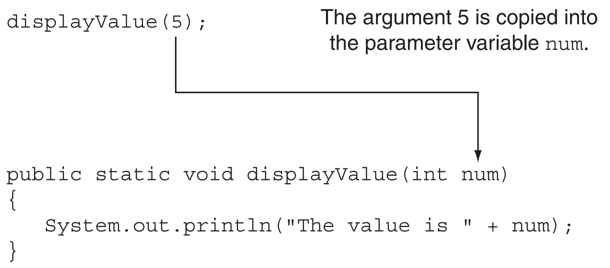
```
public static void displayValue(int num)
{
  System.out.println("The value is " + num);
}
```

Notice the integer variable declaration that appears inside the parentheses `(int num)`. This is the declaration of a parameter variable, which enables the `displayValue` method to accept an integer value as an argument. Here is an example of a call to the `displayValue` method, passing 5 as an argument:

```
displayValue(5);
```

This statement executes the `displayValue` method. The argument that is listed inside the parentheses is copied into the method's parameter variable, `num`. This is illustrated in Figure 5-7.

**Figure 5-7** Passing 5 to the `displayValue` method



```
displayValue(5);
```

The argument 5 is copied into the parameter variable `num`.

```
public static void displayValue(int num)
{
   System.out.println("The value is " + num);
}
```

Inside the `displayValue` method, the variable `num` will contain the value of whatever argument was passed into it. If we pass 5 as the argument, the method will display as follows:

```
The value is 5
```

You may also pass the contents of variables and the values of expressions as arguments. For example, the following statements call the `displayValue` method with various arguments passed:

```
displayValue(x);
displayValue(x * 4);
displayValue(Integer.parseInt("700"));
```

The first statement is simple. It passes the value in the variable `x` as the argument to the `displayValue` method. The second statement is also simple, but it does a little more work: It passes the result of the expression `x * 4` as the argument to the `displayValue` method. The third statement does even more work. It passes the value returned from the `Integer.parseInt` method as the argument to the `displayValue` method. (The `Integer.parseInt` method is called first, and its return value is passed to the `displayValue` method.) The program in Code Listing 5-5 demonstrates these method calls.

**Code Listing *5-5*** (`PassArg.java`)

```
/**
  This program demonstrates a method with a parameter.
*/

public class PassArg
{
   public static void main(String[] args)
   {
      int x = 10;

      System.out.println("I am passing values to displayValue.");
      displayValue(5);                                 // Pass 5
      displayValue(x);                                 // Pass 10
      displayValue(x * 4);                             // Pass 40
      displayValue(Integer.parseInt("700"));           // Pass 700
      System.out.println("That's all!");
   }

   /**
      The displayValue method displays the value
      of its integer parameter.
   */

   public static void displayValue(int num)
   {
      System.out.println("The value is " + num);
   }
}
```

**Program Output**

```
I am passing values to displayValue.
The value is 5
The value is 10
The value is 40
The value is 700
That's all!
```

**WARNING!** When passing a variable as an argument, simply write the variable name inside the parentheses of the method call. Do not write the data type of the argument variable in the method call. For example, the following statement will cause an error:

```
displayValue(int x);     // Error!
```

The method call should appear as follows:

```
displayValue(x);         // Correct
```

**NOTE:** In this text, the values that are passed into a method are called arguments, and the variables that receive those values are called parameters. There are several variations of these terms in use. In some circles these terms are switched in meaning. Also, some call the arguments *actual parameters* and call the parameters *formal parameters*. Others use the terms *actual argument* and *formal argument*. Regardless of which set of terms you use, it is important to be consistent.

## Argument and Parameter Data Type Compatibility

When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type. Java will automatically perform a widening conversion if the argument's data type is ranked lower than the parameter variable's data type. For example, the `displayValue` method has an `int` parameter variable. Both of the following code segments will work because the `short` and `byte` arguments are automatically converted to an `int`:

```
short s = 1;
displayValue(s);   // Converts short to int

byte b = 2;
displayValue(b);   // Converts byte to int
```

However, Java will not automatically convert an argument to a lower-ranking data type. This means that a `long`, `float`, or `double` value cannot be passed to a method that has an `int` parameter variable. For example, the following code will cause a compiler error:

```
double d = 1.0;
displayValue(d);  // Error! Can't convert double to int.
```

**TIP:** You can use a cast operator to convert a value manually to a lower-ranking data type. For example, the following code will compile:

```
double d = 1.0;
displayValue((int)d);     // This will work.
```

## Parameter Variable Scope

Recall from Chapter 2 that a variable's scope is the part of the program where the variable may be accessed by its name. A variable is visible only to statements inside the variable's scope. A parameter variable's scope is the method in which the parameter is declared. No statement outside the method can access the parameter variable by its name.

## Passing Multiple Arguments

Often it is useful to pass more than one argument to a method. Here is a method that accepts two arguments:

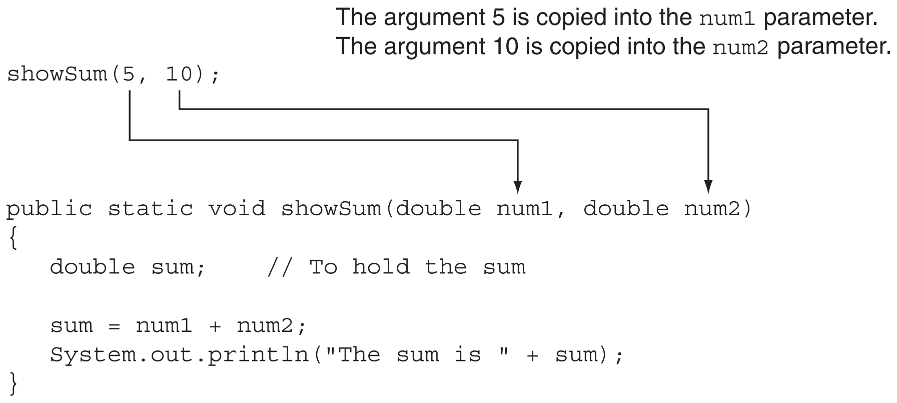
```
public static void showSum(double num1, double num2)
{
   double sum;  // To hold the sum

   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Notice that two parameter variables, `num1` and `num2`, are declared inside the parentheses in the method header. This is often referred to as a *parameter list*. Also notice that a comma separates the declarations. Here is an example of a statement that calls the method:

```
showSum(5, 10);
```

This statement passes the arguments 5 and 10 into the method. The arguments are passed into the parameter variables in the order that they appear in the method call. In other words, the first argument is passed into the first parameter variable, the second argument is passed into the second parameter variable, and so forth. So, this statement causes 5 to be passed into the `num1` parameter and 10 to be passed into the `num2` parameter. This is illustrated in Figure 5-8.

**Figure 5-8** Multiple arguments passed into multiple parameters



The argument 5 is copied into the `num1` parameter.
The argument 10 is copied into the `num2` parameter.

```
showSum(5, 10);


public static void showSum(double num1, double num2)
{
   double sum;     // To hold the sum

   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Suppose we were to reverse the order in which the arguments are listed in the method call, as shown here:

```
showSum(10, 5);
```

This would cause 10 to be passed into the `num1` parameter and 5 to be passed into the `num2` parameter. The following code segment shows one more example. This time we are passing variables as arguments.

```
double value1 = 2.5;
double value2 = 3.5;
showSum(value1, value2);
```

When the `showSum` methods executes as a result of this code, the `num1` parameter will contain 2.5 and the `num2` parameter will contain 3.5.

**WARNING!** Each parameter variable in a parameter list must have a data type listed before its name. For example, a compiler error would occur if the parameter list for the showSum method were defined as shown in the following header:

```
public static void showSum(double num1, num2) // Error!
```

A data type for both the `num1` and `num2` parameter variables must be listed, as shown here:

```
public static void showSum(double num1, double num2)
```

See the program *TwoArgs.java* in this chapter's source code folder for a complete program that demonstrates the `showSum` method. You can download the book's source code from www.pearsonhighered.com/gaddis.

## Arguments Are Passed by Value

In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable. A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If a parameter variable is changed inside a method, it has no effect on the original argument. For example, look at the program in Code Listing 5-6.

**Code Listing 5-6** (PassByValue.java)

```
/**
  This program demonstrates that only a copy of an argument
  is passed into a method.
*/

public class PassByValue
{
   public static void main(String[] args)
   {
      int number = 99; // number starts with 99

      // Display the value in number.
      System.out.println("number is " + number);

      // Call changeMe, passing the value in number
      // as an argument.
      changeMe(number);

      // Display the value in number again.
      System.out.println("number is " + number);
   }

   /**
      The changeMe method accepts an argument and then
      changes the value of the parameter.
   */

   public static void changeMe(int myValue)
   {
      System.out.println("I am changing the value.");

      // Change the myValue parameter variable to 0.
      myValue = 0;

      // Display the value in myValue.
      System.out.println("Now the value is " + myValue);
   }
}
```

**Program Output**

```
number is 99
I am changing the value.
Now the value is 0
number is 99
```

Even though the parameter variable `myValue` is changed in the `changeMe` method, the argument `number` is not modified. The `myValue` variable contains only a copy of the `number` variable.

## Passing Object References to a Method

So far you've seen examples of methods that accept primitive values as arguments. You can also write methods that accept references to objects as arguments. For example, look at the following method:

```
public static void showLength(String str)
{
  System.out.println(str + " is " + str.length() +
                     " characters long.");
}
```

This method accepts a `String` object reference as its argument, and displays a message showing the number of characters in the object. The following code shows an example of how to call the method:

```
String name = "Warren";
showLength(name);
```

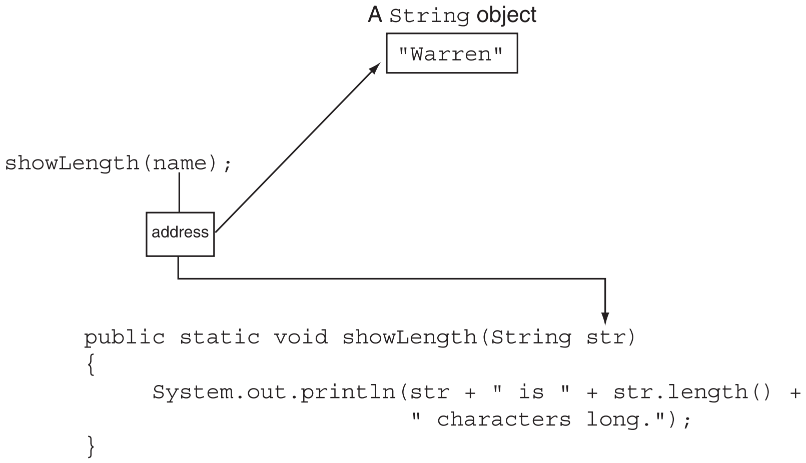
When this code executes, the `showLength` method will display the following:

```
Warren is 6 characters long.
```

When an object, such as a `String`, is passed as an argument, it is actually a reference to the object that is passed. In this example code, the `name` variable is a `String` reference variable. It is passed as an argument to the `showLength` method. The `showLength` method has a parameter variable, `str`, which is also a `String` reference variable, that receives the argument.

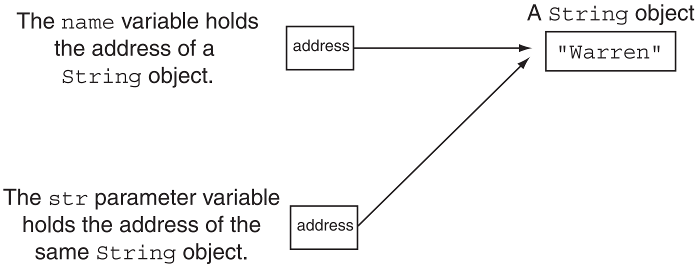
Recall that a reference variable holds the memory address of an object. When the `showLength` method is called, the address that is stored in `name` is passed into the `str` parameter variable. This is illustrated in Figure 5-9. This means that when the `showLength` method is executing, both `name` and `str` reference the same object. This is illustrated in Figure 5-10.

**Figure 5-9** Passing a reference as an argument


A `String` object
"Warren"
address

```
showLength(name);

public static void showLength(String str)
{
      System.out.println(str + " is " + str.length() +
                             " characters long.");
}
```

**Figure 5-10** Both name and `str` reference the same object


The `name` variable holds the address of a `String` object.
address
A `String` object
"Warren"
The `str` parameter variable holds the address of the same `String` object.
address

This might lead you to the conclusion that a method can change the contents of any `String` object that has been passed to it as an argument. After all, the parameter variable references the same object as the argument. However, `String` objects in Java are *immutable*, which means that they cannot be changed. For example, look at the program in Code Listing 5-7. It passes a `String` object to a method, which appears to change the object. In reality, the object is not changed.

**Code Listing 5-7** (`PassString.java`)

```
/**
  This program demonstrates that String arguments
  cannot be changed.
*/

public class PassString
{
   public static void main(String[] args)
```

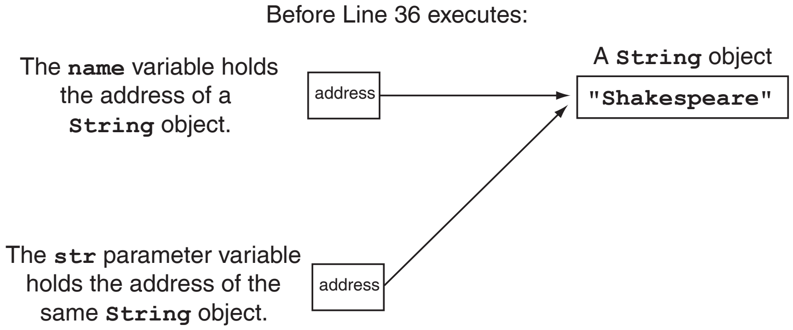
```
 9    {
10       // Create a String object containing "Shakespeare".
11       // The name variable references the object.
12       String name = "Shakespeare";
13
14       // Display the String referenced by the name variable.
15       System.out.println("In main, the name is " +
16                          name);
17
18       // Call the changeName method, passing the
19       // contents of the name variable as an argument.
20       changeName(name);
21
22       // Display the String referenced by the name variable.
23       System.out.println("Back in main, the name is " +
24                          name);
25    }
26
27    /**
28       The changeName method accepts a String as its argument
29       and assigns the str parameter to a new String.
30    */
31
32    public static void changeName(String str)
33    {
34       // Create a String object containing "Dickens".
35       // Assign its address to the str parameter variable.
36       str = "Dickens";
37
38       // Display the String referenced by str.
39       System.out.println("In changeName, the name " +
40                          "is now " + str);
41    }
42 }
```

**Program Output**

```
In main, the name is Shakespeare
In changeName, the name is now Dickens
Back in main, the name is Shakespeare
```

Let's take a closer look at this program. After line 12 executes, the `name` variable references a `String` object containing `"Shakespeare"`. In line 20 the `changeName` method is called and the `name` variable is passed as an argument. This passes the address of the `String` object into the `str` parameter variable. At this point, both `name` and `str` reference the same object, as shown in Figure 5-11.

**Figure 5-11** Before line 36 executes, both name and `str` reference the same object

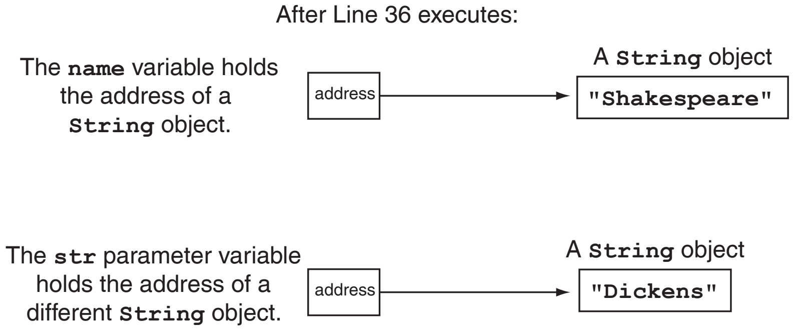

Before Line 36 executes:
The **name** variable holds the address of a **String** object.
address
A **String** object
"Shakespeare"
The **str** parameter variable holds the address of the same **String** object.
address

In the `changeName` method, line 36 executes as follows:

```
str = "Dickens";
```

At first, you might think that this statement changes the `String` object's contents to `"Dickens"`. What actually happens is that a new `String` object containing `"Dickens"` is created and its address is stored in the `str` variable. After this statement executes, the `name` variable and the `str` parameter variable reference different objects. This is shown in Figure 5-12.

**Figure 5-12** After line 36 executes, name and `str` reference different objects


After Line 36 executes:
The **name** variable holds the address of a **String** object.
address
A **String** object
"Shakespeare"
The **str** parameter variable holds the address of a different **String** object.
address
A **String** object
"Dickens"

In Chapter 9 we will discuss the immutability of `String` objects in greater detail. Until then, just remember the following point: `String` objects cannot be changed. Any time you use the = operator to assign a string literal to a `String` reference variable, a new `String` object is created in memory.

## Using the `@param` Tag in Documentation Comments

When writing the documentation comments for a method, you can provide a description of each parameter by using a `@param` tag. When the `javadoc` utility sees a `@param` tag inside a method's documentation comments, it knows that the documentation for a parameter variable appears next. The file *TwoArgs2.java*, in this chapter's source code (available at www.pearsonhighered.com/gaddis), has the following method, which uses `@param` tags in its documentation comments:

```
/**
   The showSum method displays the sum of two numbers.
   @param num1 The first number.
   @param num2 The second number.
*/

public static void showSum(double num1, double num2)
{
  double sum;  // To hold the sum

  sum = num1 + num2;
  System.out.println("The sum is " + sum);
}
```

The general format of a `@param` tag comment is as follows:

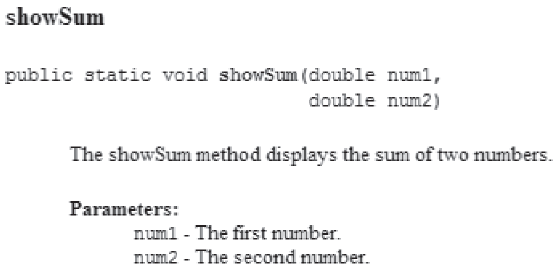
```
@param  parameterName Description
```

In the general format, `parameterName` is the name of the parameter and `Description` is a description of the parameter. Remember the following points about `@param` tag comments:

- All `@param` tags in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

When a method's documentation comments contain one or more `@param` tags, the `javadoc` utility will create a Parameters section in the method's documentation. This is where the descriptions of the method's parameters will be listed. Figure 5-13 shows the documentation generated by `javadoc` for the `showSum` method in the *TwoArgs2.java* file.

**Figure 5-13** Documentation for the showSum method in *TwoArgs2.java*



**showSum**

```
public static void showSum(double num1,
                           double num2)
```

The showSum method displays the sum of two numbers.

**Parameters:**
`num1` - The first number.
`num2` - The second number.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

5.6 What is the difference between an argument and a parameter?

5.7 Look at the following method header:

```
public static void myMethod(int num)
```

Which of the following calls to the method will cause a compiler error?

a) `myMethod(7);`

b) `myMethod(6.2);`

c)
```
long x = 99;
myMethod(x);
```

d)
```
short s = 2;
myMethod(s);
```

5.8 Suppose a method named `showValues` accepts two `int` arguments. Which of the following method headers is written correctly?

a) `public static void showValues()`

b) `public static void showValues(int num1, num2)`

c) `public static void showValues(num1, num2)`

d) `public static void showValues(int num1, int num2)`

5.9 In Java, method arguments are passed by value. What does this mean?

5.10 What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      int num1 = 99;
      double num2 = 1.5;

      System.out.println(num1 + " " + num2);
      myMethod(num1, num2);
      System.out.println(num1 + " " + num2);
   }

   public static void myMethod(int i, double d)
   {
      System.out.println(i + " " + d);
      i = 0;
      d = 0.0;
      System.out.println(i + " " + d);
   }
}
```

## 5.3 More about Local Variables

**CONCEPT: A local variable is declared inside a method and is not accessible to statements outside the method. Different methods can have local variables with the same names because the methods cannot see each other's local variables.**

In Chapter 2 we introduced the concept of local variables, which are variables that are declared inside a method. They are called *local* because they are local to the method in which they are declared. Statements outside a method cannot access that method's local variables.

Because a method's local variables are hidden from other methods, the other methods may have their own local variables with the same name. For example, look at the program in Code Listing 5-8. In addition to the `main` method, this program has two other methods: `texas` and `california`. These two methods each have a local variable named `birds`.

**Code Listing 5-8** (`LocalVars.java`)

```
/**
   This program demonstrates that two methods may have
   local variables with the same name.
*/

public class LocalVars
{
   public static void main(String[] args)
   {
      texas();
      california();
   }

   /**
      The texas method has a local variable named birds.
   */

   public static void texas()
   {
      int birds = 5000;

      System.out.println("In texas there are " +
                         birds + " birds.");
   }

   /**
      The california method also has a local variable named birds.
   */
   public static void california()
   {
      int birds = 3500;

      System.out.println("In california there are " +
                         birds + " birds.");
   }
}
```

**Program Output**

```
In texas there are 5000 birds.
In california there are 3500 birds.
```

Although there are two variables named `birds`, the program can see only one of them at a time because they are in different methods. When the `texas` method is executing, the `birds` variable declared inside `texas` is visible. When the `california` method is executing, the `birds` variable declared inside `california` is visible.

## Local Variable Lifetime

A method's local variables exist only while the method is executing. This is known as the *lifetime* of a local variable. When the method begins, its local variables and its parameter variables are created in memory, and when the method ends, the local variables and parameter variables are destroyed. This means that any value stored in a local variable is lost between calls to the method in which the variable is declared.

## Initializing Local Variables with Parameter Values

It is possible to use a parameter variable to initialize a local variable. Sometimes this simplifies the code in a method. For example, recall the following `showSum` method we discussed earlier:

```
public static void showSum(double num1, double num2)
{
   double sum;  // To hold the sum

   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

In the body of the method, the `sum` variable is declared and then a separate assignment statement assigns `num1 + num2` to `sum`. We can combine these statements into one, as shown in the following modified version of the method.

```
public static void showSum(double num1, double num2)
{
   double sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Because the scope of a parameter variable is the entire method in which it is declared, we can use parameter variables to initialize local variables.

**WARNING!** Local variables are not automatically initialized with a default value. They must be given a value before they can be used. If you attempt to use a local variable before it has been given a value, a compiler error will result. For example, look at the following method:

```
public static void myMethod()
{
   int x;
   System.out.println(x);  //Error! x has no value.
}
```

This code will cause a compiler error because the variable `x` has not been given a value, and it is being used as an argument to the `System.out.println` method.

## 5.4 Returning a Value from a Method

**CONCEPT: A method may send a value back to the statement that called the method.**

You've seen that data may be passed into a method by way of parameter variables. Data may also be returned from a method, back to the statement that called it. Methods that return a value are appropriately known as *value-returning methods*.

**VideoNote**
Returning a Value from a Method

You are already experienced at using value-returning methods. For instance, you have used the wrapper class parse methods, such as `Integer.parseInt`. Here is an example:

```
int num;
num = Integer.parseInt(“700");
```

The second line in this code calls the `Integer.parseInt` method, passing `"700"` as the argument. The method returns the integer value 700, which is assigned to the `num` variable by the = operator. You have also seen the `Math.pow` method, which returns a value. Here is an example:

```
double x;
x = Math.pow(4.0, 2.0);
```

The second line in this code calls the `Math.pow` method, passing 4.0 and 2.0 as arguments. The method calculates the value of 4.0 raised to the power of 2.0 and returns that value. The value, which is 16.0, is assigned to the `x` variable by the = operator.

In this section, we will discuss how you can write your own value-returning methods.

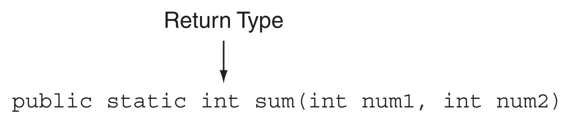
### Defining a Value-Returning Method

When you are writing a value-returning method, you must decide what type of value the method will return. This is because you must specify the data type of the return value in the method header. Recall that a `void` method, which does not return a value, uses the key word `void` as its return type in the method header. A value-returning method will use `int`, `double`, `boolean`, or any other valid data type in its header. Here is an example of a method that returns an `int` value:

```
public static int sum(int num1, int num2)
{
  int result;

  result = num1 + num2;
  return result;
}
```

The name of this method is `sum`. Notice in the method header that the return type is `int`, as shown in Figure 5-14.

**Figure 5-14** Return type in the method header



```
public static int sum(int num1, int num2)
```

This code defines a method named `sum` that accepts two `int` arguments. The arguments are passed into the parameter variables `num1` and `num2`. Inside the method, a local variable, `result`, is declared. The parameter variables `num1` and `num2` are added, and their sum is assigned to the `result` variable. The last statement in the method is as follows:

```
return result;
```

This is a `return` statement. You must have a `return` statement in a value-returning method. It causes the method to end execution and it returns a value to the statement that called the method. In a value-returning method, the general format of the `return` statement is as follows:

```
return  Expression;
```

*`Expression`* is the value to be returned. It can be any expression that has a value, such as a variable, literal, or mathematical expression. In this case, the `sum` method returns the value in the `result` variable. However, we could have eliminated the `result` variable and returned the expression `num1 + num2`, as shown in the following code:

```
public static int sum(int num1, int num2)
{
   return num1 + num2;
}
```

> **NOTE:** The `return` statement's expression must be of the same data type as the return type specified in the method header, or compatible with it. Otherwise, a compiler error will occur. Java will automatically widen the value of the `return` expression, if necessary, but it will not automatically narrow it.

## Calling a Value-Returning Method

The program in Code Listing 5-9 shows an example of how to call the `sum` method. Notice that the documentation comments for the `sum` method have a new tag, `@return`. This tag will be explained later.

**Code Listing 5-9** `(ValueReturn.java)`

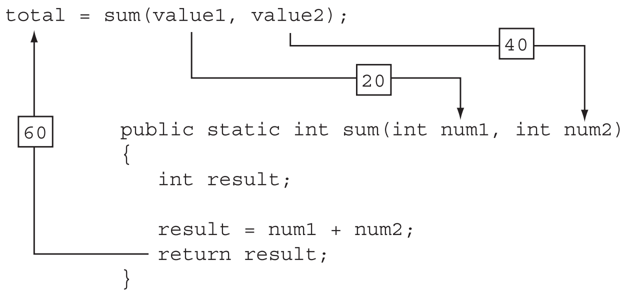
```
/**
   This program demonstrates a value-returning method.
*/

public class ValueReturn
{
   public static void main(String[] args)
   {
      int total, value1 = 20, value2 = 40;

      // Call the sum method, passing the contents of
      // value1 and value2 as arguments. Assign the
13          // return value to the total variable.
14          total = sum(value1, value2);
15 
16          // Display the contents of all these variables.
17          System.out.println("The sum of " + value1 +
18                             " and " + value2 + " is " +
19                             total);
20       }
21 
22       /**
23          The sum method returns the sum of its two parameters.
24          @param num1 The first number to be added.
25          @param num2 The second number to be added.
26          @return The sum of num1 and num2.
27       */
28 
29       public static int sum(int num1, int num2)
30       {
31          int result; // result is a local variable
32 
33          // Assign the value of num1 + num2 to result.
34          result = num1 + num2;
35 
36          // Return the value in the result variable.
37          return result;
38       }
39    }
```

**Program Output**

```
The sum of 20 and 40 is 60
```

The statement in line 14 calls the `sum` method, passing `value1` and `value2` as arguments. It assigns the value returned by the `sum` method to the `total` variable. In this case, the method will return 60. Figure 5-15 shows how the arguments are passed into the method and how a value is passed back from the method.

**Figure 5-15** Arguments passed to sum and a value returned



```
total = sum(value1, value2);

public static int sum(int num1, int num2)
{
   int result;

   result = num1 + num2;
   return result;
}
```

When you call a value-returning method, you usually want to do something meaningful with the value it returns. The *ValueReturn.java* program shows a method's return value being assigned to a variable. This is commonly how return values are used, but you can do many other things with them. For example, the following code shows a math expression that uses a call to the `sum` method:

```
int x = 10, y = 15;
double average;
average = sum(x, y) / 2.0;
```

In the last statement, the `sum` method is called with `x` and `y` as its arguments. The method's return value, which is 25, is divided by 2.0. The result, 12.5, is assigned to `average`. Here is another example:

```
int x = 10, y = 15;
System.out.println("The sum is " + sum(x, y));
```

This code sends the `sum` method's return value to `System.out.println`, so it can be displayed on the screen. The message "The sum is 25" will be displayed.

Remember, a value-returning method returns a value of a specific data type. You can use the method's return value anywhere that you can use a regular value of the same data type. This means that anywhere an `int` value can be used, a call to an `int` value-returning method can be used. Likewise, anywhere a `double` value can be used, a call to a `double` value-returning method can be used. The same is true for all other data types.

## Using the `@return` Tag in Documentation Comments

When writing the documentation comments for a value-returning method, you can provide a description of the return value by using a `@return` tag. When the `javadoc` utility sees a `@return` tag inside a method's documentation comments, it knows that a description of the method's return value appears next.

The general format of a `@return` tag comment is as follows:

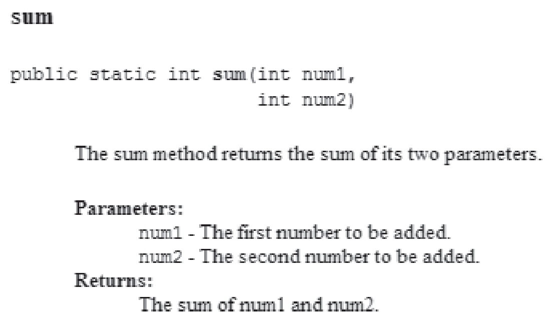
```
@return Description
```

`Description` is a description of the return value. Remember the following points about `@return` tag comments:

- The `@return` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

When a method's documentation comments contain a `@return` tag, the `javadoc` utility will create a Returns section in the method's documentation. This is where the description of the method's return value will be listed. Figure 5-16 shows the documentation generated by `javadoc` for the `sum` method in the *ValueReturn.java* file.

**Figure 5-16** Documentation for the `sum` method in `ValueReturn.java`



**sum**

```
public static int sum(int num1,
                      int num2)
```

The sum method returns the sum of its two parameters.

**Parameters:**
num1 - The first number to be added.
num2 - The second number to be added.
**Returns:**
The sum of num1 and num2.

## In the Spotlight:
## Using Methods

Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. *Get the number of cups from the user.*
2. *Convert the number of cups to fluid ounces.*
3. *Display the result.*

This algorithm lists the top level of tasks that the program needs to perform, and becomes the basis of the class's `main` method. The class will also have the following methods:

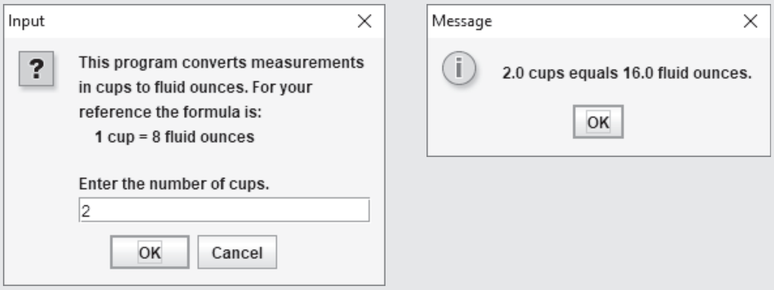
- `getCups`—This method will prompt the user to enter the number of cups, and then return that value as a `double`.
- `cupsToOunces`—This method will accept the number of cups as an argument and then return the equivalent number of fluid ounces as a `double`.
- `displayResults`—This method displays a message indicating the results of the conversion.

Code Listing 5-10 shows the program. Figure 5-17 shows interaction with the program during execution.

**Code Listing 5-10** (`CupConverter.java`)

```
import javax.swing.JOptionPane;

/**
   This program converts cups to fluid ounces.
*/

public class CupConverter
{
   public static void main(String[] args)
   {
      double cups;   // To hold the number of cups
      double ounces; // To hold the number of ounces

      // Get the number of cups.
      cups = getCups();

      // Convert the cups to fluid ounces.
      ounces = cupsToOunces(cups);

      // Display the results.
      displayResults(cups, ounces);
      System.exit(0);
   }

   /**
      The getCups method prompts the user to enter a number
      of cups.
      @return The number of cups entered by the user.
   */

   public static double getCups()
   {
      String input;    // To hold input.
      double numCups;  // To hold cups.

      // Get the number of cups from the user.
      input = JOptionPane.showInputDialog(
            "This program converts measurements\n" +
            "in cups to fluid ounces. For your\n" +
            "reference the formula is:\n" +
            "     1 cup = 8 fluid ounces\n\n" +
            "Enter the number of cups.");

      // Convert the input to a double.
      numCups = Double.parseDouble(input);

      // Return the number of cups.
      return numCups;
   }

   /**
      The cupsToOunces method converts a number of
      cups to fluid ounces, using the formula
      1 cup = 8 fluid ounces.
      @param numCups The number of cups to convert.
      @return The number of ounces.
   */

   public static double cupsToOunces(double numCups)
   {
      return numCups * 8.0;
   }

   /**
      The displayResults method displays a message showing
      the results of the conversion.
      @param cups A number of cups.
      @param ounces A number of ounces.
   */

   public static void displayResults(double cups, double ounces)
   {
      // Display the number of ounces.
      JOptionPane.showMessageDialog(null,
            cups + " cups equals " +
            ounces + " fluid ounces.");
   }
}
```

**Figure 5-17** Interaction with the `CupConverter` program (Oracle Corporate Counsel)



This input dialog appears first.
The user enters 2 and clicks OK.

This message dialog shows
the results.

This program converts measurements
in cups to fluid ounces. For your
reference the formula is:
1 cup = 8 fluid ounces

Enter the number of cups.

2

2.0 cups equals 16.0 fluid ounces.

## Returning a `boolean` Value

Frequently there is a need for a method that tests an argument and returns a `true` or `false` value indicating whether or not a condition exists. Such a method would return a `boolean` value. For example, the following method accepts an argument and returns `true` if the argument is within the range of 1 through 100, or `false` otherwise:

```
public static boolean isValid(int number)
{
   boolean status;

   if (number >= 1 && number <= 100)
      status = true;
   else
      status = false;
   return status;
}
```

The following code shows an `if-else` statement that uses a call to the method:

```
int value = 20;
if (isValid(value))
    System.out.println("The value is within range.");
else
    System.out.println("The value is out of range.");
```

When this code executes, the message “The value is within range.” will be displayed.

## Returning a Reference to an Object

A value-returning method can also return a reference to a non-primitive type, such as a `String` object. The program in Code Listing 5-11 shows such an example.

**Code Listing 5-11** (ReturnString.java)

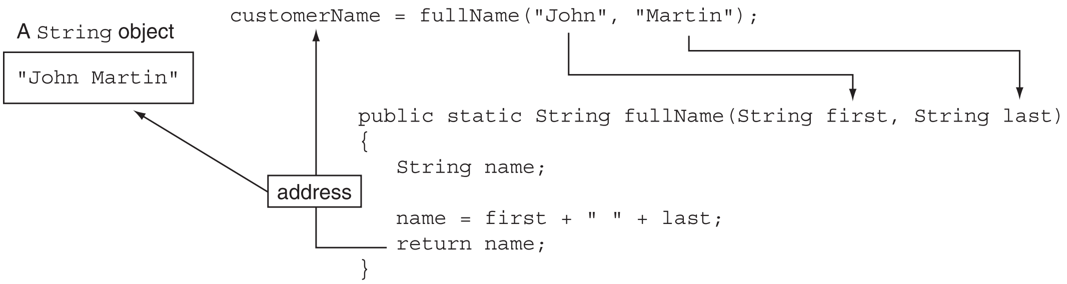
```
/**
    This program demonstrates a method that
    returns a reference to a String object.
*/

public class ReturnString
{
    public static void main(String[] args)
    {
       String customerName;

       customerName = fullName("John", "Martin");
       System.out.println(customerName);
    }

    /**
17         The fullName method accepts two String arguments
18         containing a first and last name. It concatenates
19         them into a single String object.
20         @param first The first name.
21         @param last The last name.
22         @return A reference to a String object containing
23                 the first and last names.
24     */
25 
26     public static String fullName(String first, String last)
27     {
28         String name;
29 
30         name = first + " " + last;
31         return name;
32     }
33 }
```

**Program Output**

```
John Martin
```

Line 12 calls the `fullName` method, passing `"John"` and `"Martin"` as arguments. The method returns a reference to a `String` object containing `"John Martin"`. The reference is assigned to the `customerName` variable. This is illustrated in Figure 5-18.

**Figure 5-18** The `fullName` method returning a reference to a `String` object



```
customerName = fullName("John", "Martin");
```

```
public static String fullName(String first, String last)
{
    String name;

    name = first + " " + last;
    return name;
}
```

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

5.11 Look at the following method header. What type of value does the method return?

```
public static double getValue(int a, float b, String c)
```

5.12 Write the header for a method named `days`. The method should return an `int` and have three `int` parameter variables: `years`, `months`, and `weeks`.

5.13 Write the header for a method named `distance`. The method should return a `double` and have two `double` parameter variables: `rate` and `time`.

5.14 Write the header for a method named `lightYears`. The method should return a `long` and have one `long` parameter variable: `miles`.

## 5.5 Problem Solving with Methods

**CONCEPT: A large, complex problem can be solved a piece at a time by methods.**

At the beginning of this chapter we introduced the idea of using methods to “divide and conquer” a problem. Often the best way to solve a complex problem is to break it down into smaller problems, and then solve the smaller problems. The process of breaking down a problem into smaller pieces is called *functional decomposition*.

In functional decomposition, instead of writing one long method that contains all of the statements necessary to solve a problem, small methods are written, which each solve a specific part of the problem. These small methods can then be executed in the desired order to solve the problem.

Let’s look at an example. The program in Code Listing 5-12 reads 30 days of sales amounts from a file, and then displays the total sales and average daily sales. Here’s a brief pseudocode model of the algorithm:

```
Ask the user to enter the name of the file.
Get the total of the sales amounts in the file.
Calculate the average daily sales.
Display the total and average daily sales.
```

The file *MonthlySales.txt*, in this chapter’s source code (available at www.pearsonhighered.com/gaddis), is used to test the program. Figure 5-19 shows interaction with the program during execution.

**Code Listing 5-12** (SalesReport.java)

```java
import java.util.Scanner;        // For the Scanner class
import java.io.*;                // For file I/O classes
import javax.swing.JOptionPane;  // For the JOptionPane class

/**
   This program opens a file containing the sales
   amounts for 30 days. It calculates and displays
   the total sales and average daily sales.
*/

public class SalesReport
{
   public static void main(String[] args) throws IOException
   {
      final int NUM_DAYS = 30;   // Number of days of sales
      String filename;           // The name of the file to open
      double totalSales;         // Total sales for period
```

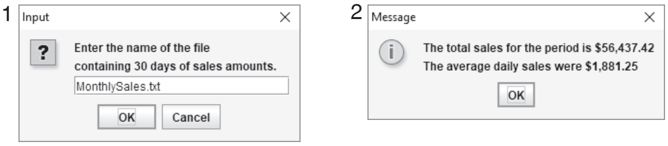
```
        double averageSales;     // Average daily sales

        // Get the name of the file.
        filename = getFileName();

        // Get the total sales from the file.
        totalSales = getTotalSales(filename);

        // Calculate the average.
        averageSales = totalSales / NUM_DAYS;

        // Display the total and average.
        displayResults(totalSales, averageSales);

        System.exit(0);
   }

   /**
        The getFileName method prompts the user to enter
        the name of the file to open.
        @return A reference to a String object containing
                the name of the file.
   */

   public static String getFileName()
   {
        String file;    // To hold the file name

        // Prompt the user to enter a file name.
        file = JOptionPane.showInputDialog(
                 "Enter the name of the file\n" +
                 "containing 30 days of sales amounts.");

        // Return the name.
        return file;
   }

   /**
        The getTotalSales method opens a file and
        reads the daily sales amounts, accumulating
        the total. The total is returned.
        @param filename The name of the file to open.
        @return The total of the sales amounts.
   */

   public static double getTotalSales(String filename) throws IOException
   {
        double total = 0.0; // Accumulator
        double sales;  // A daily sales amount

        // Open the file.
        File file = new File(filename);
        Scanner inputFile = new Scanner(file);

        // This loop processes the lines read from the file,
        // until the end of the file is encountered.
        while (inputFile.hasNext())
        {
            // Read a double from the file.
            sales = inputFile.nextDouble();

            // Add sales to the value already in total.
            total += sales;
        }

        // Close the file.
        inputFile.close();

        // Return the total sales.
        return total;
   }

   /**
        The displayResults method displays the total and
        average daily sales.
        @param total The total sales.
        @param avg The average daily sales.
   */

   public static void displayResults(double total, double avg)
   {
        // Display the formatted total and average sales.
        JOptionPane.showMessageDialog(null,
           String.format("The total sales for the period is $%,.2f\n" +
                         "The average daily sales were $%,.2f",
                         total, avg));
   }
}
```

Instead of writing the entire program in the `main` method, the algorithm was broken down into the following methods:

- `getFileName`—This method displays an input dialog box asking the user to enter the name of the file containing 30 days of sales amounts. The method returns a reference to a `String` object containing the name entered by the user.

- `getTotalSales`—This method accepts the name of a file as an argument. The file is opened, the sales amounts are read from it, and the total of the sales amounts is accumulated. The method returns the total as a `double`.
- `displayResults`—This method accepts as arguments the total sales and the average daily sales. It displays a message dialog box indicating these values.

**Figure 5-19** Interaction with the `SalesReport.java` program (Oracle Corporate Counsel)



Enter the name of the file
containing 30 days of sales amounts.
MonthlySales.txt

The total sales for the period is $56,437.42
The average daily sales were $1,881.25

## Calling Methods That Throw Exceptions

One last thing about the *SalesReport.java* program should be discussed. Notice that the `main` method header (in line 13) and the `getTotalSales` method header (in line 63) both have a `throws IOException` clause. The `getTotalSales` method has the clause because it uses a `Scanner` object to open a file. As you know from Chapter 4, any method that uses a `Scanner` object to open a file should have a throws `IOException` clause in its header. Let's quickly review why this is so.

When a `Scanner` object has a problem opening a file, it throws an exception known as `IOException`. Java requires that either (a) the exception is handled in the method that caused it to occur, or (b) the method terminates and throws the exception again. For now you must write your methods to throw the exception again because you will not learn how to handle exceptions until Chapter 11. By writing a `throws IOException` clause in a method's header, you are telling the compiler that the method does not handle the exception. Instead, it throws the exception again.

That explains why the `getTotalSales` method has the throws `IOException` clause, but it doesn't explain why the `main` method has one. After all, `main` doesn't use a `Scanner` object to perform any file operations. The reason `main` has to have the clause is because `main` calls the `getTotalSales` method. If the `Scanner` object in `getTotalSales` throws an `IOException`, the `getTotalSales` method terminates and throws the `IOException` again. That means that `main` must either handle the exception, or terminate and throw it once again. When the `main` method throws the exception, the JVM displays an error message on the screen.

**TIP:** Until you learn how to handle exceptions in Chapter 11, just remember this when writing programs that throw exceptions: If a method calls another method that has a `throws` clause in its header, then the calling method should have the same `throws` clause.

## 5.6 Common Errors to Avoid

- **Putting a semicolon at the end of a method header.** Method headers are never terminated with a semicolon.
- **Writing modifiers or return types in a method call statement.** Method modifiers and return types are written in method headers, but never in method calls.

- **Forgetting to write the empty parentheses in a call to a method that accepts no arguments.** You must always write the parentheses in a method call statement, even if the method doesn't accept arguments.
- **Forgetting to pass arguments to methods that require them.** If a method has parameter variables, you must provide arguments when calling the method.
- **Passing an argument of a data type that cannot be automatically converted to the data type of the parameter variable.** Java will automatically perform a widening conversion if the argument's data type is ranked lower than the parameter variable's data type. But Java will not automatically convert an argument to a lower-ranking data type.
- **Attempting to access a parameter variable with code outside the method where the variable is declared.** A parameter variable is visible only within the method it is declared in.
- **Not writing the data type of each parameter variable in a method header.** Each parameter variable declaration inside the parentheses of a method header must include the variable's data type.
- **Changing the contents of a method's parameter variable and expecting the argument that was passed into the parameter to change as well.** Method arguments are passed by value, which means that a copy of the argument is passed into a parameter variable. Changes to the parameter variable have no effect on the argument.
- **Using a variable to receive a method's return value when the variable's data type is incompatible with the data type of the return value.** A variable that receives a method's return value must be of a data type that is compatible with the data type of the return value.
- **Not writing a `return` statement in a value-returning method.** If a method's return type is anything other than `void`, it should return a value.
- **Not writing a required `throws` clause in a method that calls another method.** Any method that calls a method with a `throws` clause in its header must either handle the potential exception or have the same `throws` clause. You will learn how to handle exceptions in Chapter 11.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This type of method does not return a value.
   a. `null`
   b. `void`
   c. empty
   d. anonymous
2. This appears at the beginning of a method definition.
   a. semicolon
   b. parentheses
   c. body
   d. header

3. The body of a method is enclosed in __________.
   a. curly braces `{ }`
   b. square brackets `[]`
   c. parentheses `()`
   d. quotation marks `""`
4. A method header can contain __________.
   a. method modifiers
   b. the method return type
   c. the method name
   d. a list of parameter declarations
   e. all of these
   f. none of these
5. A value that is passed into a method when it is called is known as a(n) __________.
   a. parameter
   b. argument
   c. signal
   d. return value
6. A variable that receives a value that is passed into a method is known as a(n) __________.
   a. parameter
   b. argument
   c. signal
   d. return value
7. This `javadoc` tag is used to document a parameter variable.
   a. `@parameter`
   b. `@param`
   c. `@paramvar`
   d. `@arg`
8. This statement causes a method to end and sends a value back to the statement that called the method.
   a. `end`
   b. `send`
   c. `exit`
   d. `return`
9. This `javadoc` tag is used to document a method's return value.
   a. `@methodreturn`
   b. `@ret`
   c. `@return`
   d. `@returnval`
10. **True or False:** You terminate a method header with a semicolon.
11. **True or False:** When passing an argument to a method, Java will automatically perform a widening conversion (convert the argument to a higher-ranking data type), if necessary.
12. **True or False:** When passing an argument to a method, Java will automatically perform a narrowing conversion (convert the argument to a lower-ranking data type), if necessary.

13. **True or False:** A parameter variable's scope is the entire program that contains the method in which the parameter is declared.
14. **True or False:** When code in a method changes the value of a parameter, it also changes the value of the argument that was passed into the parameter.
15. **True or False:** When an object, such as a `String`, is passed as an argument, it is actually a reference to the object that is passed.
16. **True or False:** The contents of a `String` object cannot be changed.
17. **True or False:** When passing multiple arguments to a method, the order in which the arguments are passed is not important.
18. **True or False:** No two methods in the same program can have a local variable with the same name.
19. **True or False:** It is possible for one method to access a local variable that is declared in another method.
20. **True or False:** You must have a `return` statement in a value-returning method.

## Find the Error

1. Find the error in the following method definition:

```
// This method has an error!
public static void sayHello();
{
   System.out.println("Hello");
}
```

2. Look at the following method header:

```
public static void showValue(int x)
```

The following code has a call to the `showValue` method. Find the error.

```
int x = 8;
showValue(int x);   // Error!
```

3. Find the error in the following method definition:

```
// This method has an error!
public static double timesTwo(double num)
{
   double result = num * 2;
}
```

4. Find the error in the following method definition:

```
// This method has an error!
public static int half(double num)
{
   double result = num / 2.0;
   return result;
}
```

## Algorithm Workbench

1. Examine the following method header, and then write an example call to the method:

   ```
   public static void doSomething(int x)
   ```

2. Here is the code for the `displayValue` method, shown earlier in this chapter:

   ```
   public static void displayValue(int num)
   {
      System.out.println("The value is " + num);
   }
   ```

   For each of the following code segments, indicate whether it will successfully compile or cause an error:

   a. `displayValue(100);`
   b. `displayValue(6.0);`
   c. `short s = 5;`
      `displayValue(s);`
   d. `long num = 1;`
      `displayValue(num);`
   e. `displayValue(6.2f);`
   f. `displayValue((int) 7.5);`

3. Look at the following method header:

   ```
   public static void myMethod(int a, int b, int c)
   ```

   Now look at the following call to `myMethod`:

   ```
   myMethod(3, 2, 1);
   ```

   When this call executes, what value will be stored in `a`? What value will be stored in `b`? What value will be stored in `c`?

4. What will the following program display?

   ```
   public class ChangeParam
   {
     public static void main(String[] args)
     {
       int x = 1;
       double y = 3.4;
       System.out.println(x + " " + y);
       changeUs(x, y);
       System.out.println(x + " " + y);


     }
     public static void changeUs(int a, double b)
     {
       a = 0;
       b = 0.0;
   ```

```
        System.out.println(a + " " + b);
      }
    }
```

5. A program contains the following method definition:

```
public static int cube(int num)
{
    return num * num * num;
}
```

   Write a statement that passes the value 4 to this method and assigns its return value to a variable named `result`.

6. A program contains the following method:

```
public static void display(int arg1, double arg2, char arg3)
{
    System.out.println("The values are " + arg1 + ", " +
                       arg2 + ", and " + arg3);
}
```

   Write a statement that calls this method and passes the following variables as arguments:

```
char initial = 'T';
int age = 25;
double income = 50000.00;
```

7. Write a method named `timesTen`. The method should accept a `double` argument, and return a `double` value that is ten times the value of the argument.
8. Write a method named `square` that accepts an integer argument and returns the square of that argument.
9. Write a method named `getName` that prompts the user to enter his or her first name, and then returns the user's input.
10. Write a method named `quartersToDollars`. The method should accept an `int` argument that is a number of quarters, and return the equivalent number of dollars as a `double`. For example, if you pass 4 as an argument, the method should return 1.0; and if you pass 7 as an argument, the method should return 1.75.

### Short Answer

1. What is the "divide and conquer" approach to problem solving?
2. What is the difference between a void method and a value-returning method?
3. What is the difference between an argument and a parameter variable?
4. Where do you declare a parameter variable?
5. Explain what is meant by the phrase "pass by value."
6. Why do local variables lose their values between calls to the method in which they are declared?

## Programming Challenges

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. `showChar` Method

Write a method named `showChar`. The method should accept two arguments: a reference to a `String` object and an integer. The integer argument is a character position within the `String`, with the first character being at position 0. When the method executes, it should display the character at that character position. Here is an example of a call to the method:

```
showChar("New York", 2);
```

In this call, the method will display the character `w` because it is in position 2. Demonstrate the method in a complete program.

### 2. Retail Price Calculator

VideoNote The Retail Price Calculator Problem

Write a program that asks the user to enter an item's wholesale cost and its markup percentage. It should then display the item's retail price. For example:

- If an item's wholesale cost is 5.00 and its markup percentage is 100 percent, then the item's retail price is 10.00.
- If an item's wholesale cost is 5.00 and its markup percentage is 50 percent, then the item's retail price is 7.50.

The program should have a method named `calculateRetail` that receives the wholesale cost and the markup percentage as arguments, and returns the retail price of the item.

### 3. Rectangle Area—Complete the Program

If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find a partially written program named *AreaRectangle.java* in this chapter's source code folder. Your job is to complete the program. When it is complete, the program will ask the user to enter the width and length of a rectangle, and then display the rectangle's area. The program calls the following methods, which have not been written:

- `getLength`—This method should ask the user to enter the rectangle's length, and then return that value as a `double`.
- `getWidth`—This method should ask the user to enter the rectangle's width, and then return that value as a `double`.
- `getArea`—This method should accept the rectangle's length and width as arguments, and return the rectangle's area. The area is calculated by multiplying the length by the width.
- `displayData`—This method should accept the rectangle's length, width, and area as arguments, and display them in an appropriate message on the screen.

### 4. Paint Job Estimator

A painting company has determined that for every 115 square feet of wall space, one gallon of paint and eight hours of labor will be required. The company charges $18.00 per hour for labor. Write a program that allows the user to enter the number of rooms to be painted and the price of the paint per gallon. It should also ask for the square feet of wall space in each room. The program should have methods that return the following data:

- The number of gallons of paint required
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

Then it should display the data on the screen.

### 5. Falling Distance

When an object is falling because of gravity, the following formula can be used to determine the distance the object falls in a specific time period:

$$d = {}^{1}\!/_{2}\, gt^2$$

The variables in the formula are as follows: $d$ is the distance in meters, $g$ is 9.8, and $t$ is the amount of time, in seconds, that the object has been falling.

Write a method named `fallingDistance` that accepts an object's falling time (in seconds) as an argument. The method should return the distance, in meters, that the object has fallen during that time interval. Demonstrate the method by calling it in a loop that passes the values 1 through 10 as arguments, and displays the return value.

### 6. Celsius Temperature Table

The formula for converting a temperature from Fahrenheit to Celsius is

$$C = \frac{5}{9}(F - 32)$$

where $F$ is the Fahrenheit temperature and $C$ is the Celsius temperature. Write a method named `celsius` that accepts a Fahrenheit temperature as an argument. The method should return the temperature, converted to Celsius. Demonstrate the method by calling it in a loop that displays a table of the Fahrenheit temperatures 0 through 20 and their Celsius equivalents.

### 7. Test Average and Grade

Write a program that asks the user to enter five test scores. The program should display a letter grade for each score and the average test score. Write the following methods in the program:

- `calcAverage`—This method should accept five test scores as arguments and return the average of the scores.
- `determineGrade`—This method should accept a test score as an argument and return a letter grade for the score, based on the following grading scale:

| Score | Letter Grade |
|---|---|
| 90–100 | A |
| 80–89 | B |
| 70–79 | C |
| 60–69 | D |
| Below 60 | F |

### 8. Conversion Program

Write a program that asks the user to enter a distance in meters. The program will then present the following menu of selections:

1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program

The program will convert the distance to kilometers, inches, or feet, depending on the user's selection. Here are the specific requirements:

- Write a `void` method named `showKilometers`, which accepts the number of meters as an argument. The method should display the argument converted to kilometers. Convert the meters to kilometers using the following formula:

  ```
  kilometers = meters * 0.001
  ```

- Write a `void` method named `showInches`, which accepts the number of meters as an argument. The method should display the argument converted to inches. Convert the meters to inches using the following formula:

  ```
  inches = meters * 39.37
  ```

- Write a `void` method named `showFeet`, which accepts the number of meters as an argument. The method should display the argument converted to feet. Convert the meters to feet using the following formula:

  ```
  feet = meters * 3.281
  ```

- Write a `void` method named `menu` that displays the menu of selections. This method should not accept any arguments.
- The program should continue to display the menu until the user enters 4 to quit the program.
- The program should not accept negative numbers for the distance in meters.
- If the user selects an invalid choice from the menu, the program should display an error message.

Here is an example session with the program, using console input. The user's input is shown in bold.

```
Enter a distance in meters: 500 [Enter]
1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program

Enter your choice: 1 [Enter]
500 meters is 0.5 kilometers.

1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program
```

```
Enter your choice: 3 [Enter]
500 meters is 1640.5 feet.

1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program

Enter your choice: 4 [Enter]
Bye!
```

### 9. Distance Traveled Modification

The distance a vehicle travels can be calculated as follows:

$$Distance = Speed * Time$$

Write a method named `distance` that accepts a vehicle's speed and time as arguments, and returns the distance the vehicle has traveled. Modify the "Distance Traveled" program you wrote in Chapter 4 (Programming Challenge 2) to use the method.

### 10. Stock Profit

The profit from the sale of a stock can be calculated as follows:

$$Profit = ((NS \times SP) - SC) - ((NS \times PP) + PC)$$

where $NS$ is the number of shares, $PP$ is the purchase price per share, $PC$ is the purchase commission paid, $SP$ is the sale price per share, and $SC$ is the sale commission paid. If the calculation yields a positive value, then the sale of the stock resulted in a profit. If the calculation yields a negative number, then the sale resulted in a loss.

Write a method that accepts as arguments the number of shares, the purchase price per share, the purchase commission paid, the sale price per share, and the sale commission paid. The method should return the profit (or loss) from the sale of stock. Demonstrate the method in a program that asks the user to enter the necessary data and displays the amount of the profit or loss.

### 11. Multiple Stock Sales

Use the method that you wrote for Programming Challenge 10 (Stock Profit) in a program that calculates the total profit or loss from the sale of multiple stocks. The program should ask the user for the number of stock sales, and the necessary data for each stock sale. It should accumulate the profit or loss for each stock sale and then display the total.

### 12. Kinetic Energy

In physics, an object that is in motion is said to have kinetic energy. The following formula can be used to determine a moving object's kinetic energy:

$$KE = {}^{1}\!/_{2}\, mv^2$$

The variables in the formula are as follows: $KE$ is the kinetic energy, $m$ is the object's mass in kilograms, and $v$ is the object's velocity, in meters per second.

Write a method named `kineticEnergy` that accepts an object's mass (in kilograms) and velocity (in meters per second) as arguments. The method should return the amount of kinetic energy that the object has. Demonstrate the method by calling it in a program that asks the user to enter values for mass and velocity.

### 13. `isPrime` Method

A prime number is a number that is evenly divisible only by itself and 1. For example, the number 5 is prime because it can be evenly divided only by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6.

Write a method named `isPrime`, which takes an integer as an argument and returns `true` if the argument is a prime number, or `false` otherwise. Demonstrate the method in a complete program.

> **TIP:** Recall that the `%` operator divides one number by another, and returns the remainder of the division. In an expression such as `num1 % num2`, the `%` operator will return 0 if `num1` is evenly divisible by `num2`.

### 14. Prime Number List

Use the `isPrime` method that you wrote in Programming Challenge 13 in a program that stores a list of all the prime numbers from 1 through 100 in a file.

### 15. Even/Odd Counter

You can use the following logic to determine whether a number is even or odd:

```
if ((number % 2) == 0)
{
  // The number is even.
}
else
{
  // The number is odd.
}
```

Write a program with a method named `isEven` that accepts an `int` argument. The method should return `true` if the argument is even, or `false` otherwise. The program's `main` method should use a loop to generate 100 random integers. It should use the `isEven` method to determine whether each random number is even, or odd. When the loop is finished, the program should display the number of even numbers that were generated, and the number of odd numbers.

### 16. Present Value

Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years, you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula, which is known as the present value formula, to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- *P* is the **present value**, or the amount that you need to deposit today.
- *F* is the **future value** that you want in the account. (In this case, *F* is $10,000.)
- *r* is the **annual interest rate**.
- *n* is the **number of years** that you plan to let the money sit in the account.

Write a method named `presentValue` that performs this calculation. The method should accept the future value, annual interest rate, and number of years as arguments. It should return the present value, which is the amount that you need to deposit today. Demonstrate the method in a program that lets the user experiment with different values for the formula's terms.

### 17. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows.

1. When the program begins, a random number in the range of 1 through 3 is generated. If the number is 1, then the computer has chosen rock. If the number is 2, then the computer has chosen paper. If the number is 3, then the computer has chosen scissors. (Don't display the computer's choice yet.)
2. The user enters his or her choice of "rock", "paper", or "scissors" at the keyboard. (You can use a menu if you prefer.)
3. The computer's choice is displayed.
4. A winner is selected according to the following rules:
    - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
    - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
    - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
    - If both players make the same choice, the game must be played again to determine the winner.

Be sure to divide the program into methods that perform each major task.

### 18. ESP Game

Write a program that tests your ESP (extrasensory perception). The program should randomly select the name of a color from the following list of words:

*Red, Green, Blue, Orange, Yellow*

To select a word, the program can generate a random number. For example, if the number is 0, the selected word is *Red*; if the number is 1, the selected word is *Green*; and so forth.

Next, the program should ask the user to enter the color that the computer has selected. After the user has entered his or her guess, the program should display the name of the randomly selected color. The program should repeat this 10 times and then display the number of times the user correctly guessed the selected color. Be sure to modularize the program into methods that perform each major task.

CHAPTER

# 6

# A First Look at Classes

**TOPICS**

6.1 Objects and Classes
6.2 Writing a Simple Class, Step by Step
6.3 Instance Fields and Methods
6.4 Constructors
6.5 Passing Objects as Arguments
6.6 Overloading Methods and Constructors
6.7 Scope of Instance Fields
6.8 Packages and `import` Statements
6.9 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities
On the Web: Case Study—The `Amortization` Class
6.10 Common Errors to Avoid

## 6.1 Objects and Classes

**CONCEPT: An object is a software component that exists in memory and serves a specific purpose in a program. An object is created from a class that contains code describing the object.**

If you have ever driven a car, you know that a car consists of a lot of components. It has a steering wheel, an accelerator pedal, a brake pedal, a gear shifter, a speedometer, and numerous other devices that the driver interacts with. There are also a lot of components under the hood, such as the engine, the battery, the radiator, and so forth. So, a car is not just one single object, but rather a collection of objects that work together.

This same notion applies to computer programming as well. Most programming languages in use today are object-oriented. With an object-oriented language, such as Java, you create programs that are made of objects. In programming, however, an object isn't a physical device, like a steering wheel or a brake pedal; it's a software component that exists in the computer's memory and performs a specific task. In software, an object has two general capabilities:

- An object can store data. The data stored in an object are commonly called *fields*.
- An object can perform operations. The operations that an object can perform are called *methods*.

Objects are very important in Java. Here are some examples of objects that you have previously learned about:

- If you need to read input from the keyboard, or from a file, you can use a `Scanner` object.
- If you need to generate random numbers, you can use a `Random` object.
- If you need to write output to a file, you can use a `PrintWriter` object.

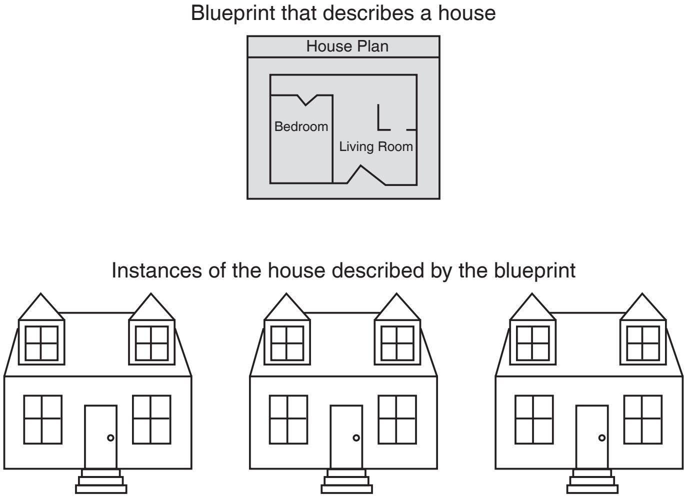
When a program needs the services of a particular type of object, it creates that object in memory, and then calls that object's methods as necessary.

## Classes: Where Objects Come From

Objects are very useful, but they don't just magically appear in your program. Before a specific type of object can be used by a program, that object has to be created in memory. And, before an object can be created in memory, you must have a class for the object.

A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods). You can think of a class as a code "blueprint" that can be used to create a particular type of object. It serves a purpose similar to that of the blueprint for a house. The blueprint itself is not a house, but rather a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 6-1.

**Figure 6-1** A blueprint and houses built from the blueprint


Blueprint that describes a house
House Plan
Bedroom
Living Room
Instances of the house described by the blueprint

So, a class is not an object, but a description of an object. When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

**NOTE:** Up to this chapter, you have used classes for a different purpose: as containers for a program's methods. All of the Java programs that you have written so far have had a class containing a `main` method, and possibly other methods. In this chapter, you will learn how to write classes from which objects can be created.

## Classes in the Java API

So far, the objects that you have used in your programs are created from classes in the Java API. For example, each time you create a `Scanner` object, you are creating an instance of a class named `Scanner`, which is in the Java API. Likewise, when you create a `Random` object, you are creating an instance of a class named `Random`, which is in the Java API. The same is true for `PrintWriter` objects. When you need to write data to a file, you create an instance of the `PrintWriter` class, which is in the Java API. Look at Code Listing 6-1, a program that uses all of these types of objects.

**Code Listing 6-1** `(ObjectDemo.java)`

```java
import java.util.Scanner; // Needed for the Scanner class
import java.util.Random;  // Needed for the Random class
import java.io.*;         // Needed for file I/O classes

/**
  This program writes random numbers to a file.
*/

public class ObjectDemo
{
   public static void main(String[] args) throws IOException
   {
      int maxNumbers;   // Max number of random numbers
      int number;       // To hold a random number

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create a Random object to generate random numbers.
      Random rand = new Random();

      // Create a PrintWriter object to open the file.
      PrintWriter outputFile = new PrintWriter("numbers.txt");

      // Get the number of random numbers to write.
      System.out.print("How many random numbers should I write? ");
      maxNumbers = keyboard.nextInt();

29          // Write the random numbers to the file.
30          for (int count = 0; count < maxNumbers; count++)
31          {
32             // Generate a random integer.
33             number = rand.nextInt();
34
35             // Write the random integer to the file.
36             outputFile.println(number);
37          }
38
39          // Close the file.
40          outputFile.close();
41          System.out.println("Done");
42       }
43    }
```

**Program Output with Example Input Shown in Bold**

```
How many random numbers should I write? 10 [Enter]
Done
```

In a nutshell, this program writes a specified number of random numbers to a file named *numbers.txt*. When the program runs, it asks the user for the number of random numbers to write. It then writes that many numbers to the file. To do its job, it creates three objects:

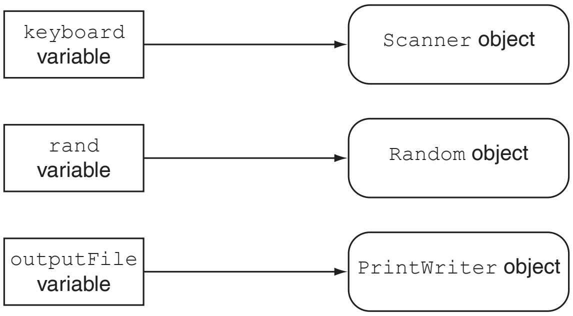
- In line 17 it creates an instance of the `Scanner` class, and assigns the object's address to a variable named `keyboard`. The object will be used to read keyboard input.
- In line 20 it creates an instance of the `Random` class, and assigns the object's address to a variable named `rand`. The object will be used to generate random numbers.
- In line 23 it creates an instance of the `PrintWriter` class, and assigns the object's address to a variable named `outputFile`. The object will be used to write output to the *numbers.txt* file.

Figure 6-2 illustrates the three objects that the program creates. As the program runs, it uses these objects to accomplish certain tasks. For example:

- In line 27 the `Scanner` object's `nextInt` method is called to read the user's input (which is the number of random numbers to generate). The value that is returned from the method is assigned to the `maxNumbers` variable.
- In line 33 the `Random` object's `nextInt` method is called to get a random integer. The value that is returned from the method is assigned to the `number` variable.
- In line 36 the `PrintWriter` object's `println` method is called to write the value of the `number` variable to the file.
- In line 40 the `PrintWriter` object's `close` method is called to close the file.

This simple example demonstrates how most programs work. A program creates the various objects that it needs to complete its job. Each object has a set of methods that can be called, causing the object to perform an operation. When the program needs an object to do something, it calls the appropriate method.

**Figure 6-2** Objects created by the `ObjectDemo` program



**NOTE:** The `import` statements that appear in lines 1 through 3 of Code Listing 6-1 make the `Scanner`, `Random`, and `PrintWriter` classes available to the program. You will learn more about how the Java API is organized, and why you need these `import` statements later in this chapter.

## Primitive Variables vs. Objects

Chapter 2 introduced you to the Java primitive data types: `byte`, `short`, `int`, `long`, `char`, `float`, `double`, and `boolean`. By now you have seen many programs that use both primitive data types and objects. In fact, the program in Code Listing 6-1 uses two primitive variables (`maxNumbers` and `number`, both `int` variables), as well as a `Scanner` object, a `Random` object, and a `PrintWriter` object.

You've probably noticed that the steps required to create an object differ from the steps required to create a primitive variable. For example, to create an `int` variable, you simply need a declaration such as the following:

```
int wholeNumber;
```

But, to create an object, you have to write some extra code. For example, the following statement creates a `Random` object:

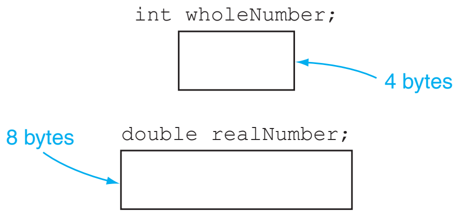
```
Random rand = new Rand();
```

Primitive variables, such as `int`s, `double`s, and so forth, are simply storage locations in the computer's memory. A primitive data type is called "primitive" because a variable created with a primitive data type has no built-in capabilities other than storing a value. When you declare a primitive variable, the compiler sets aside, or allocates, a chunk of memory that is big enough for that variable. For example, look at the following variable declarations:

```
int wholeNumber;
double realNumber;
```

Recall from Chapter 2 that an `int` uses 4 bytes of memory and a `double` uses 8 bytes of memory. These declaration statements will cause memory to be allocated as shown in Figure 6-3.

**Figure 6-3** Memory allocation

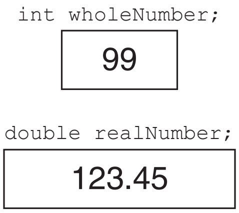


The memory that is allocated for a primitive variable is the actual location that will hold any value that is assigned to that variable. For example, suppose we use the following statements to assign values to the variables shown in Figure 6-3:

```
wholeNumber = 99;
realNumber = 123.45;
```

Figure 6-4 shows how the assigned values are stored in each variable's memory location.

**Figure 6-4** Values assigned to the variables



As you can see from these illustrations, primitive variables are very straightforward. When you are working with a primitive variable, you are using a storage location that holds a piece of data.

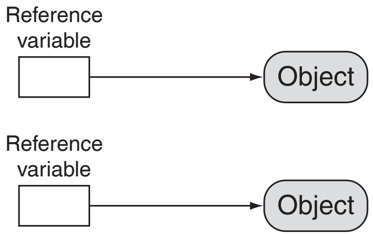
This is different from the way that objects work. When you are working with an object, you are typically using two things:

- The object itself, which must be created in memory
- A reference variable that refers to the object

The object that is created in memory holds data of some sort and performs operations of some sort. (Exactly what the data and operations are depends on what kind of object it is.) In order to work with the object in code, you need some way to refer to the object. That's where the reference variable comes in. The reference variable doesn't hold an actual piece of data that your program will work with. Instead, it holds the object's memory address. We say that the variable references the object. When you want to work with the object, you use the variable that references it.

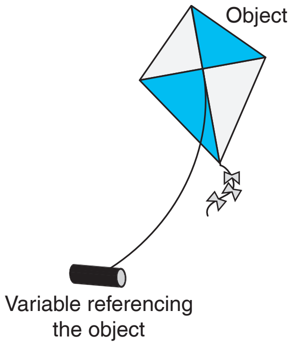
Reference variables, also known as class type variables, can be used only to reference objects. Figure 6-5 illustrates two objects that have been created in memory, each referenced by a variable.

**Figure 6-5** Two objects referenced by variables



To understand how reference variables and objects work together, think about flying a kite. In order to fly a kite, you need a spool of string attached to it. When the kite is airborne, you use the spool of string to hold on to the kite and control it. This is similar to the relationship between an object and the variable that references the object. As shown in Figure 6-6, the object is like the kite, and the variable that references the object is like the spool of string.

**Figure 6-6** The kite and string metaphor



Creating an object typically requires the following two steps:

1. You declare a reference variable.
2. You create the object in memory, and assign its memory address to the reference variable.

After you have performed these steps, you can use the reference variable to work with the object. Once again, here is the familiar example of how you create an object from the `Random` class:

```
Random rand = new Random();
```

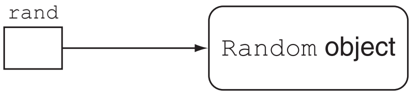
Let's look at the different parts of this statement:

- The first part of the statement, appearing on the left side of the `=` operator, reads `Random rand`. This declares a variable named `rand`, which can be used to reference an object of the `Random` type.
- The second part of the statement, appearing on the right side of the `=` operator, reads `new Random()`. The `new` operator creates an object in memory, and returns that object's memory address. So, the expression `new Random()` creates an object from the `Random` class, and returns that object's memory address.

- The `=` operator assigns the memory address that was returned from the `new` operator to the `rand` variable.

After this statement executes, the `rand` variable will reference a `Random` object, as shown in Figure 6-7. The `rand` variable can then be used to perform operations with the object, such as generating random numbers.

**Figure 6-7** The `rand` variable references a `Random` object



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

6.1 What does an object use its fields for?

6.2 What are an object's methods?

6.3 How is a class like a blueprint?

6.4 You have programs that create `Scanner`, `Random`, and `PrintWriter` objects. Where are the `Scanner`, `Random`, and `PrintWriter` classes?

6.5 What does the `new` operator do?

6.6 What values do reference variables hold?

6.7 How is the relationship between an object and a reference variable similar to a kite and a spool of string?

## 6.2 Writing a Simple Class, Step by Step

**CONCEPT: You can write your own classes to create the objects that you need in a program. We will go through the process of writing a class in a step-by-step fashion.**

The Java API provides many prewritten classes, ready for use in your programs. Sometimes, however, you will wish you had an object to perform a specific task, and no such class will exist in the Java API. This is not a problem, because you can write your own classes with the specific fields and methods that you need for any situation.

In this section we will write a class named `Rectangle`. Each object that is created from the `Rectangle` class will be able to hold data about a rectangle. Specifically, a `Rectangle` object will have the following fields:

- `length`. The `length` field will hold the rectangle's length.
- `width`. The `width` field will hold the rectangle's width.

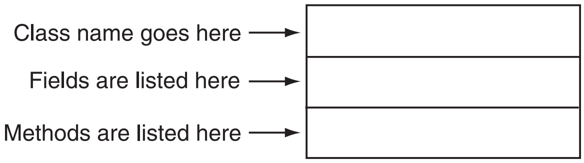
The `Rectangle` class will also have the following methods:

- `setLength`. The `setLength` method will store a value in an object's `length` field.
- `setWidth`. The `setWidth` method will store a value in an object's `width` field.

- `getLength`. The `getLength` method will return the value in an object's `length` field.
- `getWidth`. The `getWidth` method will return the value in an object's `width` field.
- `getArea`. The `getArea` method will return the area of the rectangle, which is the result of an object's length multiplied by its width.

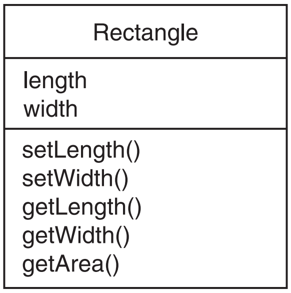
When designing a class it is often helpful to draw a UML diagram. UML stands for Unified Modeling Language. It provides a set of standard diagrams for graphically depicting object-oriented systems. Figure 6-8 shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's fields. The bottom section holds a list of the class's methods.

**Figure 6-8** General layout of a UML diagram for a class



Following this layout, Figure 6-9 shows a UML diagram for our `Rectangle` class. Throughout this book we frequently use UML diagrams to illustrate classes.

**Figure 6-9** UML diagram for the Rectangle class



### Writing the Code for a Class

Now that we have identified the fields and methods that we want the `Rectangle` class to have, let's write the Java code. First, we use an editor to create a new file named *Rectangle.java*. In the *Rectangle.java* file we will start by writing a general class "skeleton" as follows:

```
public class Rectangle
{
}
```

**VideoNote**
Writing Classes and
Creating Objects

The key word `public`, which appears in the first line, is an access specifier. An access specifier indicates how the class may be accessed. The `public` access specifier indicates that the class will be publicly available to code that is written outside the *Rectangle.java* file. Almost all of the classes that we write in this book are `public`.

Following the access specifier is the key word `class`, followed by `Rectangle`, which is the name of the class. On the next line an opening brace appears, which is followed by a closing brace. The contents of the class, which are the fields and methods, will be written inside these braces. The general format of a class definition is as follows:

```
AccessSpecifier class Name
{
    Members
}
```

In general terms, the fields and methods that belong to a class are referred to as the class's *members*.

### Writing the Code for the Class Fields

Let's continue writing our `Rectangle` class by filling in the code for some of its members. First we will write the code for the class's two fields, `length` and `width`. We will use variables of the `double` data type for the fields. The new lines of code are shown in bold, as follows:

```
public class Rectangle
{
     private double length;
     private double width;
}
```

These two lines of code that we have added declare the variables `length` and `width`. Notice that both declarations begin with the key word `private`, preceding the data type. The key word `private` is an access specifier. It indicates that these variables may not be accessed by statements outside the class.

By using the `private` access modifier, a class can hide its data from code outside the class. When a class's fields are hidden from outside code, the data is protected from accidental corruption. It is a common practice to make all of a class's fields `private` and to provide access to those fields through methods only. In other words, a class usually has `private` fields, and `public` methods that access those fields. Table 6-1 summarizes the difference between the `private` and `public` access specifiers.

**Table 6-1** Summary of the `private` and `public` access specifiers for class members

| Access Specifier | Description |
|---|---|
| `private` | When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class. |
| `public` | When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside. |

### Writing the `setLength` Method

Now we will begin writing the class methods. We will start with the `setLength` method. This method will allow code outside the class to store a value in the `length` field.

Code Listing 6-2 shows the `Rectangle` class at this stage of its development. The `setLength` method is in lines 17 through 20. (This file is in the source code folder *Chapter 06\Rectangle Class Phase 1*.)

**Code Listing 6-2** (`Rectangle.java`)

```
 1  /**
 2     Rectangle class, phase 1
 3     Under construction!
 4  */
 5
 6  public class Rectangle
 7  {
 8     private double length;
 9     private double width;
10
11     /**
12        The setLength method stores a value in the
13        length field.
14        @param len The value to store in length.
15     */
16
17     public void setLength(double len)
18     {
19        length = len;
20     }
21  }
```

In lines 11 through 15, we write a block comment that gives a brief description of the method. It's important always to write comments that describe a class's methods so that in the future, anyone reading the code will understand it. The definition of the method appears in lines 17 through 20. Here is the method header:

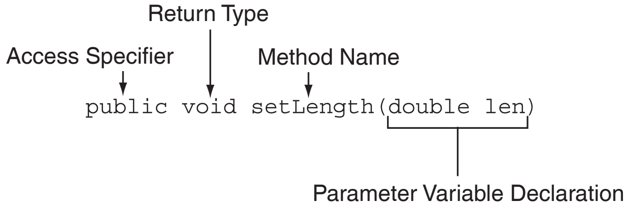
```
public void setLength(double len)
```

The method header looks very much like any other method header that you learned to write in Chapter 5. Let's look at the parts as follows:

- `public`. The key word `public` is an access specifier. It indicates that the method may be called by statements outside the class.
- `void`. This is the method's return type. The key word `void` indicates that the method returns no data to the statement that called it.
- `setLength`. This is the name of the method.
- `(double len)`. This is the declaration of a parameter variable of the `double` data type, named `len`.

Figure 6-10 labels each part of the header for the `setLength` method.

**Figure 6-10** Header for the `setLength` method


Return Type
Access Specifier
Method Name
public void setLength(double len)
Parameter Variable Declaration

Notice that the word `static` does not appear in the method header. When a method is designed to work on an instance of a class, it is referred to as an *instance method*, and you do not write the word `static` in the header. Because this method will store a value in the `length` field of an instance of the `Rectangle` class, it is an instance method. We will discuss this in greater detail later.

After the header, the body of the method appears inside a set of braces:

```
{
     length = len;
}
```

The body of this method has only one statement, which assigns the value of `len` to the `length` field. When the method executes, the `len` parameter variable will hold the value of an argument that is passed to the method. That value is assigned to the `length` field.

Before adding the other methods to the class, it might help if we demonstrate how the `setLength` method works. First, notice that the `Rectangle` class does not have a `main` method. This class is not a complete program, but is a blueprint that `Rectangle` objects may be created from. Other programs will use the `Rectangle` class to create objects. The programs that create and use these objects will have their own `main` methods. We can demonstrate the class's `setLength` method by saving the current contents of the *Rectangle.java* file and then creating the program shown in Code Listing 6-3.

**Code Listing 6-3** (`LengthDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   setLength method.
*/

public class LengthDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object and assign its
      // address to the box variable.
12        Rectangle box = new Rectangle();
13
14        // Indicate what we are doing.
15        System.out.println("Sending the value 10.0 " +
16                           "to the setLength method.");
17
18        // Call the box object's setLength method.
19        box.setLength(10.0);
20
21        // Indicate we are done.
22        System.out.println("Done.");
23     }
24  }
```

**Program Output**

```
Sending the value 10.0 to the setLength method.
Done.
```

The program in Code Listing 6-3 must be saved as *LengthDemo.java* in the same folder or directory as the file *Rectangle.java*. The following command can then be used with the JDK to compile the program:

```
javac LengthDemo.java
```

When the compiler reads the source code for *LengthDemo.java* and sees that a class named `Rectangle` is being used, it looks in the current folder or directory for the file *Rectangle.class*. That file does not exist, however, because we have not yet compiled *Rectangle.java*. So, the compiler searches for the file *Rectangle.java* and compiles it. This creates the file *Rectangle.class*, which makes the `Rectangle` class available. The compiler then finishes compiling *LengthDemo.java*. The resulting *LengthDemo.class* file may be executed with the following command:

```
java LengthDemo
```

The output of the program is shown at the bottom of Code Listing 6-3.

Let's look at each statement in this program's `main` method. First, the program uses the following statement, in line 12, to create a `Rectangle` object and associate it with a variable:

```
Rectangle box = new Rectangle();
```

Let's dissect the statement into two parts. The first part of the statement,

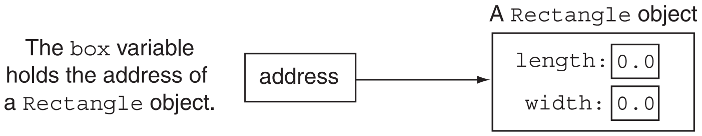
```
Rectangle box
```

declares a variable named `box`. The data type of the variable is `Rectangle`. (Because the word `Rectangle` is not the name of a primitive data type, Java assumes it to be the name of a class.) Recall that a variable of a class type is a reference variable, and it holds the memory address of an object. When a reference variable holds an object's memory address, it is said that the variable references the object. So, the variable `box` will be used to reference a `Rectangle` object. The second part of the statement is as follows:

```
= new Rectangle();
```

This part of the statement uses the key word `new`, which creates an object in memory. After the word `new`, the name of a class followed by a set of parentheses appears. This specifies the class that the object should be created from. In this case, an object of the `Rectangle` class is created. The memory address of the object is then assigned (by the = operator) to the variable `box`. After the statement executes, the variable `box` will reference the object that was created in memory. This is illustrated in Figure 6-11.

**Figure 6-11** The box variable references a `Rectangle` class object


A Rectangle object
The box variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0

Notice that Figure 6-11 shows the `Rectangle` object's `length` and `width` fields set to 0. All of a class's numeric fields are initialized to 0 by default.

**TIP:** The parentheses in this statement are required. It would be an error to write the statement as follows:

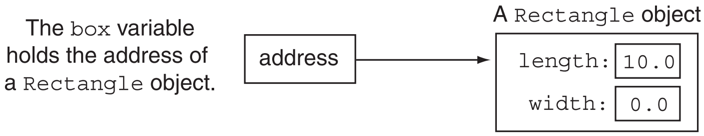
```
Rectangle box = new Rectangle; // ERROR!!
```

The statement in lines 15 and 16 uses the `System.out.println` method to display a message on the screen. The next statement, in line 19, calls the `box` object's `setLength` method as follows:

```
box.setLength(10.0);
```

This statement passes the argument 10.0 to the `setLength` method. When the method executes, the value 10.0 is copied into the `len` parameter variable. The method assigns the value of `len` to the `length` field and then terminates. Figure 6-12 shows the state of the `box` object after the method executes.

**Figure 6-12** The state of the box object after the `setLength` method executes


A Rectangle object
The box variable holds the address of a Rectangle object.
address
length: 10.0
width: 0.0

### Writing the `setWidth` Method

Now that we've seen how the `setLength` method works, let's add the `setWidth` method to the `Rectangle` class. The `setWidth` method is similar to `setLength`. It accepts an argument, which is assigned to the `width` field. Code Listing 6-4 shows the updated `Rectangle` class. The `setWidth` method is in lines 28 through 31. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 2*.)

**Code Listing 6-4** (`Rectangle.java`)

```
 1  /**
 2     Rectangle class, phase 2
 3     Under construction!
 4  */
 5
 6  public class Rectangle
 7  {
 8     private double length;
 9     private double width;
10
11     /**
12        The setLength method stores a value in the
13        length field.
14        @param len The value to store in length.
15     */
16
17     public void setLength(double len)
18     {
19        length = len;
20     }
21
22     /**
23        The setWidth method stores a value in the
24        width field.
25        @param w The value to store in width.
26     */
27
28     public void setWidth(double w)
29     {
30        width = w;
31     }
32  }
```

The `setWidth` method has a parameter variable named `w`, which is assigned to the `width` field. For example, assume that `box` references a `Rectangle` object and the following statement is executed:

```
box.setWidth(20.0);
```

After this statement executes, the `box` object's `width` field will be set to 20.0.

### Writing the `getLength` and `getWidth` Methods

Because the `length` and `width` fields are private, we wrote the `setLength` and `setWidth` methods to allow code outside the `Rectangle` class to store values in the fields. We must also write methods that allow code outside the class to get the values that are stored in these fields. That's what the `getLength` and `getWidth` methods will do. The `getLength` method will return the value stored in the `length` field, and the `getWidth` method will return the value stored in the `width` field.

Here is the code for the `getLength` method:

```
public double getLength()
{
    return length;
}
```

Assume that `size` is a `double` variable and that `box` references a `Rectangle` object, and the following statement is executed:

```
size = box.getLength();
```

This statement assigns the value that is returned from the `getLength` method to the `size` variable. After this statement executes, the `size` variable will contain the same value as the `box` object's `length` field.

The `getWidth` method is similar to `getLength`. The code for the method follows:

```
public double getWidth()
{
     return width;
}
```

This method returns the value that is stored in the `width` field. For example, assume that `size` is a `double` variable and that `box` references a `Rectangle` object, and the following statement is executed:

```
size = box.getWidth();
```

This statement assigns the value that is returned from the `getWidth` method to the `size` variable. After this statement executes, the `size` variable will contain the same value as the `box` object's `width` field.

Code Listing 6-5 shows the `Rectangle` class with all of the members we have discussed so far. The code for the `getLength` and `getWidth` methods is shown in lines 33 through 53. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 3*.)

**Code Listing 6-5** (`Rectangle.java`)

```
/**
   Rectangle class, phase 3
   Under construction!
*/

public class Rectangle
{
   private double length;
   private double width;

   /**
      The setLength method stores a value in the
      length field.
      @param len The value to store in length.
   */

   public void setLength(double len)
   {
      length = len;
   }

   /**
      The setWidth method stores a value in the
      width field.
      @param w The value to store in width.
   */

   public void setWidth(double w)
   {
      width = w;
   }

   /**
      The getLength method returns a Rectangle
      object's length.
      @return The value in the length field.
   */

   public double getLength()
   {
      return length;
   }

   /**
```

```
      The getWidth method returns a Rectangle
      object's width.
      @return The value in the width field.
   */

   public double getWidth()
   {
      return width;
   }
}
```

Before continuing we should demonstrate how these methods work. Look at the program in Code Listing 6-6. (This file is also stored in the source code folder *Chapter 06\Rectangle Class Phase 3*.)

**Code Listing 6-6** (`LengthWidthDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   setLength, setWidth, getLength, and getWidth methods.
*/

public class LengthWidthDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle();

      // Call the object's setLength method, passing 10.0
      // as an argument.
      box.setLength(10.0);

      // Call the object's setWidth method, passing 20.0
      // as an argument.
      box.setWidth(20.0);

      // Display the object's length and width.
      System.out.println("The box's length is " +
                         box.getLength());
      System.out.println("The box's width is " +
                         box.getWidth());
   }
}
```

**Program Output**

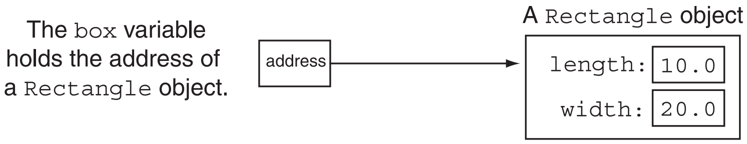
```
The box's length is 10.0
The box's width is 20.0
```

Let's take a closer look at the program. In line 11, this program creates a `Rectangle` object, which is referenced by the `box` variable. Then the following statements execute in lines 15 and 19:

```
box.setLength(10.0);
box.setWidth(20.0);
```

After these statements execute, the `box` object's `length` field is set to 10.0 and its `width` field is set to 20.0. The state of the object is shown in Figure 6-13.

**Figure 6-13** State of the box object



Next, the following statement in lines 22 and 23 executes as follows:

```
System.out.println("The box's length is " +
                   box.getLength());
```

This statement calls the `box.getLength()` method, which returns the value 10.0. The following message is displayed on the screen:

```
The box's length is 10.0
```

Then the following statement executes in lines 24 and 25:

```
System.out.println("The box's width is " +
                   box.getWidth());
```

This statement calls the `box.getWidth()` method, which returns the value 20.0. The following message is displayed on the screen:

```
The box's width is 20.0
```

### Writing the getArea Method

The last method we will write for the `Rectangle` class is `getArea`. This method returns the area of a rectangle, which is its length multiplied by its width. Here is the code for the `getArea` method:

```
public double getArea()
{
    return length * width;
}
```

This method returns the result of the mathematical expression `length * width`. For example, assume that `area` is a `double` variable and that `box` references a `Rectangle` object, and the following code is executed:

```
box.setLength(10.0);
box.setWidth(20.0);
area = box.getArea();
```

The last statement assigns the value that is returned from the `getArea` method to the `area` variable. After this statement executes, the `area` variable will contain the value 200.0.

Code Listing 6-7 shows the `Rectangle` class with all of the members we have discussed so far. The `getArea` method appears in lines 61 through 64. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 4*.)

**Code Listing 6-7** `(Rectangle.java)`

```
 1  /**
 2     Rectangle class, phase 4
 3     Under construction!
 4  */
 5
 6  public class Rectangle
 7  {
 8     private double length;
 9     private double width;
10
11     /**
12        The setLength method stores a value in the
13        length field.
14        @param len The value to store in length.
15     */
16
17     public void setLength(double len)
18     {
19        length = len;
20     }
21
22     /**
23        The setWidth method stores a value in the
24        width field.
25        @param w The value to store in width.
26     */
27
28     public void setWidth(double w)
29     {
30        width = w;
31     }
32
33     /**
34        The getLength method returns a Rectangle
35        object's length.
36        @return The value in the length field.
37     */
38
```

```
   public double getLength()
   {
      return length;
   }

   /**
      The getWidth method returns a Rectangle
      object's width.
      @return The value in the width field.
   */

   public double getWidth()
   {
      return width;
   }

   /**
      The getArea method returns a Rectangle
      object's area.
      @return The product of length times width.
   */

   public double getArea()
   {
      return length * width;
   }
}
```

The program in Code Listing 6-8 demonstrates all the methods of the `Rectangle` class, including `getArea`. (This file is also stored in the source code folder *Chapter 06\Rectangle Class Phase 4*.)

**Code Listing 6-8** (`RectangleDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   setLength, setWidth, getLength, getWidth, and
   getArea methods.
*/

public class RectangleDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle();

      // Set length to 10.0 and width to 20.0.
      box.setLength(10.0);
      box.setWidth(20.0);

      // Display the length.
      System.out.println("The box's length is " +
                         box.getLength());

      // Display the width.
      System.out.println("The box's width is " +
                         box.getWidth());

      // Display the area.
      System.out.println("The box's area is " +
                         box.getArea());
   }
}
```

**Program Output**

```
The box's length is 10.0
The box's width is 20.0
The box's area is 200.0
```

## Accessor and Mutator Methods

As mentioned earlier, it is common practice to make all of a class's fields private and to provide public methods for accessing and changing those fields. This ensures that the object owning those fields is in control of all changes being made to them. A method that gets a value from a class's field but does not change it is known as an *accessor method*. A method that stores a value in a field or changes the value of a field in some other way is known as a *mutator method*. In the `Rectangle` class, the methods `getLength` and `getWidth` are accessors, and the methods `setLength` and `setWidth` are mutators.

**NOTE:** Mutator methods are sometimes called "setters" and accessor methods are sometimes called "getters."

## The Importance of Data Hiding

*Data hiding* is an important concept in object-oriented programming. An object hides its internal data from code that is outside the class that the object is an instance of. Only the class's methods may directly access and make changes to the object's internal data. You hide an object's internal data by making the class's fields `private`, and making the methods that access those fields `public`.

As a beginning student, you might be wondering why you would want to hide the data that is inside the classes you create. As you learn to program, you will be the user of your own classes, so it might seem that you are putting forth a great effort to hide data from yourself. If you write software in industry, however, the classes that you create will be used as components in large software systems, and programmers other than yourself will be using your classes. By hiding a class's data, and allowing it to be accessed only through the class's methods, you can better ensure that the class will operate as you intended it to.

## Avoiding Stale Data

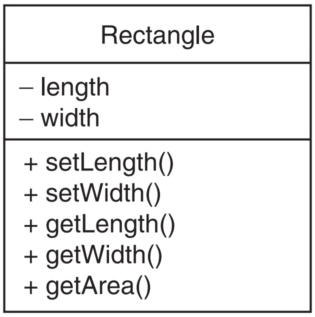
In the `Rectangle` class, the `getLength` and `getWidth` methods return the values stored in fields, but the `getArea` method returns the result of a calculation. You might be wondering why the area of the rectangle is not stored in a field, like the length and the width. The area is not stored in a field because it could potentially become stale. When the value of an item is dependent on other data and that item is not updated when the other data is changed, it is said that the item has become *stale*. If the area of the rectangle were stored in a field, the value of the field would become incorrect as soon as either the `length` or `width` field changed.

When designing a class, you should take care not to store in a field calculated data that can potentially become stale. Instead, provide a method that returns the result of the calculation.

## Showing Access Specification in UML Diagrams

In Figure 6-9 we presented a UML diagram for the `Rectangle` class. The diagram listed all of the members of the class but did not indicate which members were private and which were public. In a UML diagram, you have the option to place a `-` character before a member name to indicate that it is private, or a `+` character to indicate that it is public. Figure 6-14 shows the UML diagram modified to include this notation.

**Figure 6-14** UML diagram for the `Rectangle` class



## Data Type and Parameter Notation in UML Diagrams

The Unified Modeling Language also provides notation that you may use to indicate the data types of fields, methods, and parameter variables. To indicate the data type of a field, place a colon followed by the name of the data type after the name of the field. For example, the `length` field in the `Rectangle` class is a `double`. It could be listed in the UML diagram as follows:

```
- length : double
```

The return type of a method can be listed in the same manner: After the method's name, place a colon followed by the return type. The `Rectangle` class's `getLength` method returns a `double`, so it could be listed in the UML diagram as follows:

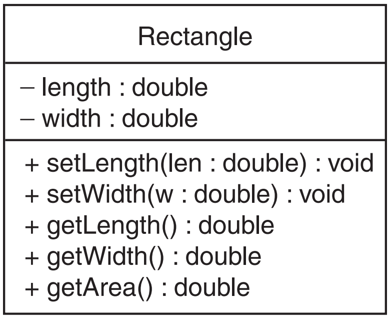
```
+ getLength() : double
```

Parameter variables and their data types may be listed inside a method's parentheses. For example, the `Rectangle` class's `setLength` method has a `double` parameter named `len`, so it could be listed in the UML diagram as follows:

```
+ setLength(len : double) : void
```

Figure 6-15 shows a UML diagram for the `Rectangle` class with parameter and data type notation.

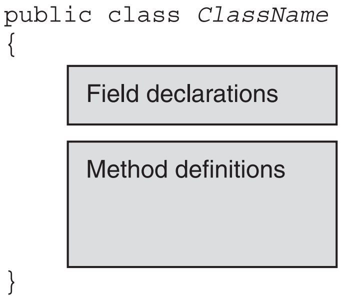
**Figure 6-15** UML diagram for the `Rectangle` class with parameter and data type notation


Rectangle
– length : double
– width : double
+ setLength(len : double) : void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double

## Layout of Class Members

Notice that in the `Rectangle` class, the field variables are declared first and then the methods are defined. You are not required to write field declarations before the method definitions. In fact, some programmers prefer to write the definitions for the public methods first and write the declarations for the private fields last. Regardless of the style you use, you should be consistent. In this book, we always write the field declarations first, followed by the method definitions. Figure 6-16 shows this layout.

**Figure 6-16** Typical layout of class members


public class *ClassName*
{
Field declarations
Method definitions
}

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

6.8 You hear someone make the following comment: "A blueprint is a design for a house. A carpenter can use the blueprint to build the house. If the carpenter wishes, he or she can build several identical houses from the same blueprint." Think of this as a metaphor for classes and objects. Does the blueprint represent a class, or does it represent an object?

6.9 In this chapter we used the metaphor of a kite attached to a spool of string to describe the relationship between an object and a reference variable. In this metaphor, does the kite represent an object, or a reference variable?

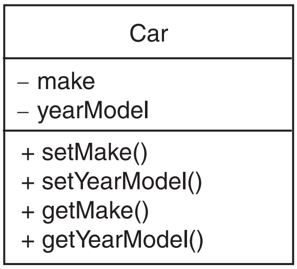
6.10 When a variable is said to reference an object, what is actually stored in the variable?

6.11 A string literal, such as `"Joe"`, causes what type of object to be created?

6.12 Look at the UML diagram in Figure 6-17 and answer the following questions:
   a) What is the name of the class?
   b) What are the fields?
   c) What are the methods?
   d) What are the private members?
   e) What are the public members?

6.13 Assume that `limo` is a variable that references an instance of the class shown in Figure 6-17. Write a statement that calls `setMake` and passes the argument `"Cadillac"`.

**Figure 6-17** UML diagram



6.14 What does the key word `new` do?

6.15 What is an accessor? What is a mutator?

6.16 What is a stale data item?

## 6.3 Instance Fields and Methods

**CONCEPT: Each instance of a class has its own set of fields, which are known as instance fields. You can create several instances of a class and store different values in each instance's fields. The methods that operate on an instance of a class are known as instance methods.**

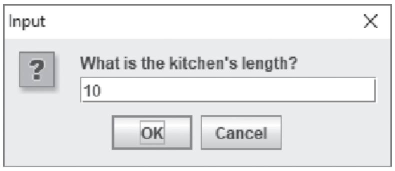
The program in Code Listing 6-8 creates one instance of the `Rectangle` class. It is possible to create many instances of the same class, each with its own data. For example, the *RoomAreas.java* program in Code Listing 6-9 creates three instances of the `Rectangle` class, referenced by the variables `kitchen`, `bedroom`, and `den`. Figure 6-18 shows example interaction with the program. (The file in Code Listing 6-9 is stored in the source code folder *Chapter 06\Rectangle Class Phase 4*.)

**Code Listing 6-9** (RoomAreas.java)

```
import javax.swing.JOptionPane;

/**
   This program creates three instances of the
   Rectangle class.
*/

public class RoomAreas
{
   public static void main(String[] args)
   {
      double number;          // To hold a number
      double totalArea;       // The total area
      String input;           // To hold user input

      // Create three Rectangle objects.
      Rectangle kitchen = new Rectangle();
      Rectangle bedroom = new Rectangle();
      Rectangle den = new Rectangle();

      // Get and store the dimensions of the kitchen.
      input = JOptionPane.showInputDialog("What is the " +
                                          "kitchen's length?");
      number = Double.parseDouble(input);
      kitchen.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "kitchen's width?");
      number = Double.parseDouble(input);
      kitchen.setWidth(number);

      // Get and store the dimensions of the bedroom.
      input = JOptionPane.showInputDialog("What is the " +
                                          "bedroom's length?");
      number = Double.parseDouble(input);
      bedroom.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "bedroom's width?");
      number = Double.parseDouble(input);
      bedroom.setWidth(number);

      // Get and store the dimensions of the den.
      input = JOptionPane.showInputDialog("What is the " +
                                          "den's length?");
      number = Double.parseDouble(input);
      den.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "den's width?");
      number = Double.parseDouble(input);
      den.setWidth(number);

      // Calculate the total area of the rooms.
      totalArea = kitchen.getArea() + bedroom.getArea()
                  + den.getArea();

      // Display the total area of the rooms.
      JOptionPane.showMessageDialog(null, "The total area " +
                                 "of the rooms is " + totalArea);

      System.exit(0);
   }
}
```

**Figure 6-18** Interaction with the `RoomAreas.java` program (Oracle Corporate Counsel)

1

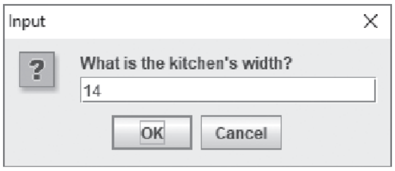


Input

What is the kitchen's length?

10

2

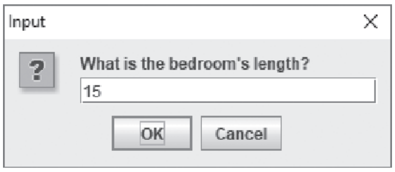


Input

What is the kitchen's width?

14

3

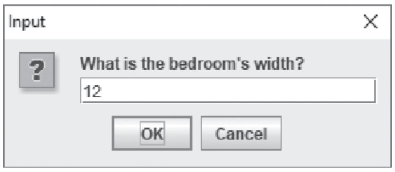


Input

What is the bedroom's length?

15

4

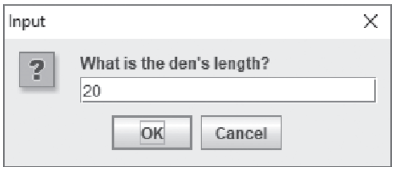


Input

What is the bedroom's width?

12

5

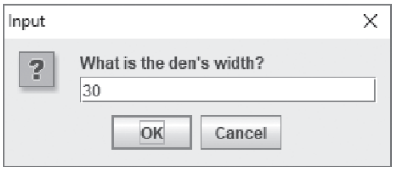


Input

What is the den's length?

20

6

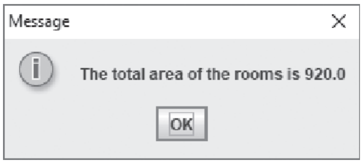


Input

What is the den's width?

30

7



Message

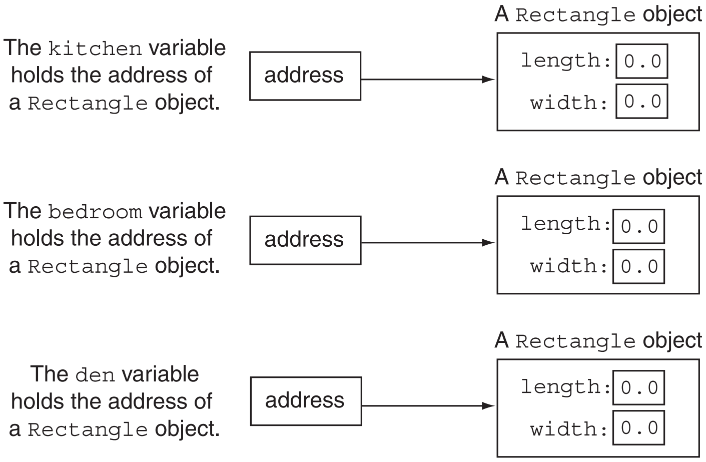
The total area of the rooms is 920.0

In lines 17, 18, and 19, the following code creates three objects, each an instance of the `Rectangle` class:

```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

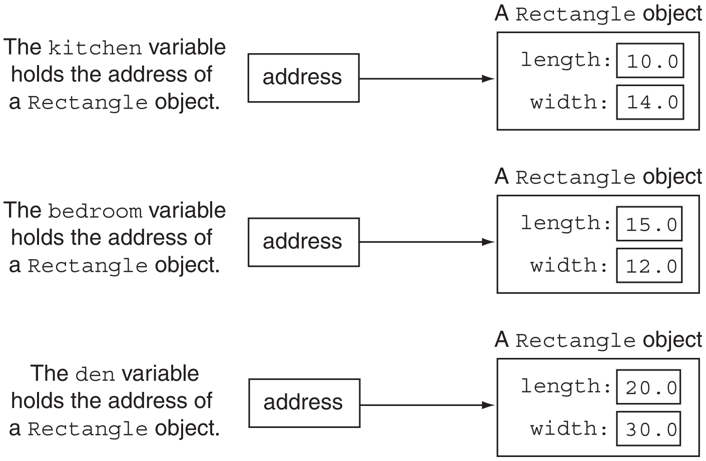
Figure 6-19 illustrates how the `kitchen`, `bedroom`, and `den` variables reference the objects.

**Figure 6-19** The `kitchen`, `bedroom`, and `den` variables reference `Rectangle` objects


A Rectangle object
The kitchen variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0
A Rectangle object
The bedroom variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0
A Rectangle object
The den variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0

In the example session with the program, the user enters 10 and 14 as the length and width of the kitchen, 15 and 12 as the length and width of the bedroom, and 20 and 30 as the length and width of the den. Figure 6-20 shows the states of the objects after these values are stored in them.

**Figure 6-20** States of the objects after data has been stored in them


A Rectangle object
The kitchen variable holds the address of a Rectangle object.
address
length: 10.0
width: 14.0
A Rectangle object
The bedroom variable holds the address of a Rectangle object.
address
length: 15.0
width: 12.0
A Rectangle object
The den variable holds the address of a Rectangle object.
address
length: 20.0
width: 30.0

Notice from Figure 6-20 that each instance of the `Rectangle` class has its own `length` and `width` variables. For this reason, the variables are known as *instance variables*, or *instance fields*. Every instance of a class has its own set of instance fields and can store its own values in those fields.

The methods that operate on an instance of a class are known as *instance methods*. All of the methods in the `Rectangle` class are instance methods because they perform operations on specific instances of the class. For example, look at the following statement in line 25 of the *RoomAreas.java* program:

```
kitchen.setLength(number);
```

This statement calls the `setLength` method, which stores a value in the `kitchen` object's `length` field. Now look at the following statement in line 35:

```
bedroom.setLength(number);
```

This statement also calls the `setLength` method, but this time it stores a value in the `bedroom` object's `length` field. Likewise, the following statement in line 45 calls the `setLength` method to store a value in the `den` object's `length` field:

```
den.setLength(number);
```

The `setLength` method stores a value in a specific instance of the `Rectangle` class. This is true of all of the methods that are members of the `Rectangle` class.

**NOTE:** As previously mentioned, instance methods do not have the key word `static` in their headers.

## Checkpoint

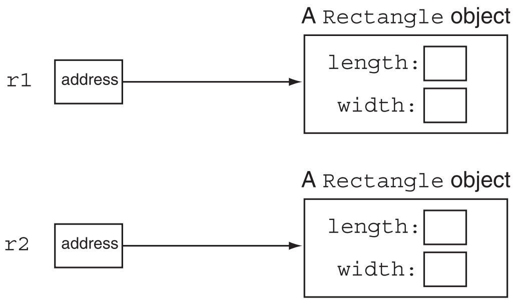
MyProgrammingLab™ *www.myprogramminglab.com*

6.17 Assume that `r1` and `r2` are variables that reference `Rectangle` objects, and the following statements are executed:

```
r1.setLength(5.0);
r2.setLength(10.0);
r1.setWidth(20.0);
r2.setWidth(15.0);
```

Fill in the boxes in Figure 6-21 that represent each object's `length` and `width` fields.

**Figure 6-21** Fill in the boxes for each field



## 6.4 Constructors

**CONCEPT: A constructor is a method that is automatically called when an object is created.**

VideoNote
Initializing an Object with a Constructor

A constructor is a method that is automatically called when an instance of a class is created. Constructors normally perform initialization or setup operations, such as storing initial values in instance fields. They are called "constructors" because they help construct an object.

A constructor method has the same name as the class. For example, Code Listing 6-10 shows the first few lines of a new version of the `Rectangle` class. In this version of the class, a constructor has been added. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 5*.)

**Code Listing 6-10** `(Rectangle.java)`

```
/**
   Rectangle class, phase 5
*/

public class Rectangle
{
   private double length;
   private double width;

   /**
      Constructor
      @param len The length of the rectangle.
      @param w The width of the rectangle.
   */

   public Rectangle(double len, double w)
   {
      length = len;
      width = w;
   }
```

. . . *The remainder of the class has not changed, and is not shown.*

This constructor accepts two arguments, which are passed into the `len` and `w` parameter variables. The parameter variables are then assigned to the `length` and `width` fields.

Notice that the constructor's header doesn't specify a return type—not even `void`. This is because constructors are not executed by explicit method calls and cannot return a value.

The method header for a constructor takes the following general format:

```
AccessSpecifier ClassName(Parameters...)
```

Here is an example statement that declares the variable `box`, creates a `Rectangle` object, and passes the values 7.0 and 14.0 to the constructor.

```
Rectangle box = new Rectangle(7.0, 14.0);
```

After this statement executes, `box` will reference a `Rectangle` object whose `length` field is set to 7 and whose `width` field is set to 14. The program in Code Listing 6-11 demonstrates the `Rectangle` class constructor. (This file is also stored in the source code folder *Chapter 06\Rectangle Class Phase 5*.)

**Code Listing 6-11** (`ConstructorDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   constructor.
*/

public class ConstructorDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object, passing 5.0 and
      // 15.0 as arguments to the constructor.
      Rectangle box = new Rectangle(5.0, 15.0);

      // Display the length.
      System.out.println("The box's length is " +
                         box.getLength());

      // Display the width.
      System.out.println("The box's width is " +
                         box.getWidth());

      // Display the area.
      System.out.println("The box's area is " +
                         box.getArea());
   }
}
```

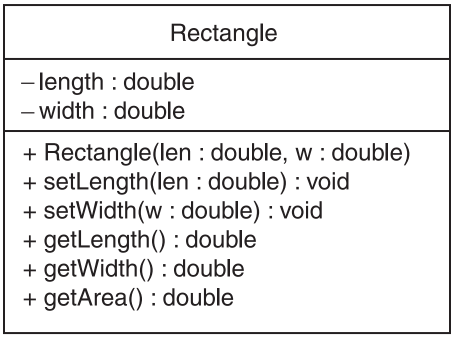
**Program Output**

```
The box's length is 5.0
The box's width is 15.0
The box's area is 75.0
```

## Showing Constructors in a UML Diagram

There is more than one accepted way of showing a class's constructor in a UML diagram. In this book, we simply show a constructor just as any other method, except we list no return type. Figure 6-22 shows a UML diagram for the `Rectangle` class with the constructor listed.

**Figure 6-22** UML diagram for the `Rectangle` class showing the constructor



## Uninitialized Local Reference Variables

The program in Code Listing 6-11 initializes the `box` variable with the address of a `Rectangle` object. Reference variables can also be declared without being initialized, as in the following statement:

```
Rectangle box;
```

Note that this statement does not create a `Rectangle` object. It only declares a variable named `box` that can be used to reference a `Rectangle` object. Because the `box` variable does not yet hold an object's address, it is an *uninitialized reference variable*.

After declaring the reference variable, the following statement can be used to assign it the address of an object. This statement creates a `Rectangle` object, passes the values 7.0 and 14.0 to its constructor, and assigns the object's address to the `box` variable:

```
box = new Rectangle(7.0, 14.0);
```

Once this statement executes, the `box` variable will reference a `Rectangle` object.

You need to be careful when using uninitialized reference variables. Recall from Chapter 5 that local variables *must* be initialized or assigned a value before they can be used. This is also true for local reference variables. A local reference variable must reference an object before it can be used. Otherwise a compiler error will occur.

## The Default Constructor

When an object is created, its constructor is *always* called. But what if we do not write a constructor in the object's class? If you do not write a constructor in a class, Java automatically provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*. The default constructor doesn't accept arguments. It sets all of the object's numeric fields to 0 and `boolean` fields to `false`. If the object has any fields that are reference variables, the default constructor sets them to the special value `null`, which means that they do not reference anything.

The *only* time that Java provides a default constructor is when you do not write your own constructor for a class. For example, at the beginning of this chapter we developed the `Rectangle` class without writing a constructor for it. When we compiled the class, the compiler generated a default constructor that set both the `length` and `width` fields to 0.0. Assume that the following code uses that version of the class to create a `Rectangle` object:

```
// We wrote no constructor for the Rectangle class.
Rectangle r = new Rectangle();  // Calls the default constructor
```

When we created `Rectangle` objects using that version of the class, we did not pass any arguments to the default constructor because the default constructor doesn't accept arguments.

Later we added our own constructor to the class. The constructor that we added accepts arguments for the `length` and `width` fields. When we compiled the class at that point, Java did not provide a default constructor. The constructor that we added became the only constructor that the class has. When we create `Rectangle` objects with that version of the class, we *must* pass the length and width arguments to the constructor. Using that version of the class, the following statement would cause an error because we have not provided arguments for the constructor:

```
// Now we wrote our own constructor for the Rectangle class.
Rectangle box = new Rectangle(); // Error! Must now pass arguments.
```

Because we have added our own constructor, which requires two arguments, the class no longer has a default constructor.

## Writing Your Own No-Arg Constructor

A constructor that does not accept arguments is known as a *no-arg constructor*. The default constructor doesn't accept arguments, so it is considered a no-arg constructor. In addition, you can write your own no-arg constructor. For example, suppose we wrote the following constructor for the `Rectangle` class:

```
public Rectangle()
{
    length = 1.0;
    width = 1.0;
}
```

If we were using this constructor in our `Rectangle` class, we would not pass any arguments when creating a `Rectangle` object. The following code shows an example. After this code executes, the `Rectangle` object's `length` and `width` fields would both be set to 1.0.

```
// Now we have written our own no-arg constructor.
Rectangle r = new Rectangle();  // Calls the no-arg constructor
```

## The `String` Class Constructor

Earlier in this chapter (in Section 6.1) we discussed the difference between creating a primitive variable and creating an object. You create primitive variables with simple declaration statements, and you create objects with the `new` operator. There is one class, however, that can be instantiated without the `new` operator: the `String` class.

Because string operations are so common, Java allows you to create `String` objects in the same way that you create primitive variables. Here is an example:

```
String name = "Joe Mahoney";
```

This statement creates a `String` object in memory, initialized with the string literal `"Joe Mahoney"`. The object is referenced by the `name` variable. If you wish, you can use the `new` operator to create a `String` object, and initialize the object by passing a string literal to the constructor, as shown here:

```
String name = new String("Joe Mahoney");
```

**NOTE:** `String` objects are a special case in Java. Because they are so commonly used, Java provides numerous shortcut operations with `String` objects that are not possible with objects of other types. In addition to creating a `String` object without using the `new` operator, you can use the `=` operator to assign values to `String` objects, the `+` operator to concatenate strings, and so forth. Chapter 9 discusses several of the `String` class methods.

## In the Spotlight:

### Creating the `CellPhone` Class

Wireless Solutions, Inc., is a business that sells cell phones and wireless service. You are a programmer in the company's information technology (IT) department, and your team is designing a program to manage all of the cell phones that are in inventory. You have been asked to design a class that represents a cell phone. The data that should be kept as fields in the class are as follows:

- The name of the phone's manufacturer will be assigned to the `manufact` field.
- The phone's model number will be assigned to the `model` field.
- The phone's retail price will be assigned to the `retailPrice` field.

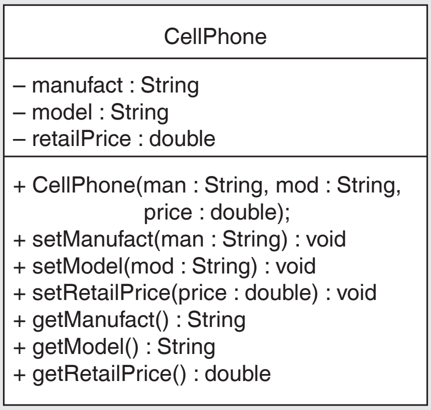
The class will also have the following methods:

- A constructor that accepts arguments for the manufacturer, model number, and retail price.
- A `setManufact` method that accepts an argument for the manufacturer. This method will allow us to change the value of the `manufact` field after the object has been created, if necessary.
- A `setModel` method that accepts an argument for the model. This method will allow us to change the value of the `model` field after the object has been created, if necessary.

- A `setRetailPrice` method that accepts an argument for the retail price. This method will allow us to change the value of the `retailPrice` field after the object has been created, if necessary.
- A `getManufact` method that returns the phone's manufacturer.
- A `getModel` method that returns the phone's model number.
- A `getRetailPrice` method that returns the phone's retail price.

Figure 6-23 shows a UML diagram for the class. Code Listing 6-12 shows the class definition.

**Figure 6-23** UML diagram for the `CellPhone` class


CellPhone
– manufact : String
– model : String
– retailPrice : double
+ CellPhone(man : String, mod : String, price : double);
+ setManufact(man : String) : void
+ setModel(mod : String) : void
+ setRetailPrice(price : double) : void
+ getManufact() : String
+ getModel() : String
+ getRetailPrice() : double

**Code Listing 6-12** `(CellPhone.java)`

```
/**
  The CellPhone class holds data about a cell phone.
*/

public class CellPhone
{
  // Fields
  private String manufact;       // Manufacturer
  private String model;          // Model
  private double retailPrice;    // Retail price

  /**
     Constructor
     @param man The phone's manufacturer.
     @param mod The phone's model number.
     @param price The phone's retail price.
  */

  public CellPhone(String man, String mod, double price)
  {
     manufact = man;
     model = mod;
```

```
      retailPrice = price;
   }

   /**
      The setManufact method sets the phone's
      manufacturer name.
      @param man The phone's manufacturer.
   */

   public void setManufact(String man)
   {
      manufact = man;
   }

   /**
      The setModel method sets the phone's
      model number.
      @param mod The phone's model number.
   */

   public void setMod(String mod)
   {
      model = mod;
   }

   /**
      The setRetailPrice method sets the phone's
      retail price.
      @param price The phone's retail price.
   */

   public void setRetailPrice(double price)
   {
      retailPrice = price;
   }

   /**
      getManufact method
      @return The name of the phone's manufacturer.
   */

   public String getManufact()
   {
      return manufact;
   }

   /**
      getModel method
      @return The phone's model number.
   */

   public String getModel()
   {
      return model;
   }

   /**
      getretailPrice method
      @return The phone's retail price.
   */

   public double getRetailPrice()
   {
      return retailPrice;
   }
}
```

The `CellPhone` class will be used by several programs that your team is developing. To perform a simple test of the class, you write the program shown in Code Listing 6-13. This is a simple program that prompts the user for the phone's manufacturer, model number, and retail price. An instance of the `CellPhone` class is created and the data is assigned to its attributes.

**Code Listing 6-13** (`CellPhoneTest.java`)

```
import java.util.Scanner;

/**
  This program runs a simple test
  of the CellPhone class.
*/

public class CellPhoneTest
{
  public static void main(String[] args)
  {
     String testMan;    // To hold a manufacturer
     String testMod;    // To hold a model number
     double testPrice;  // To hold a price

     // Create a Scanner object for keyboard input.
     Scanner keyboard = new Scanner(System.in);

     // Get the manufacturer name.
     System.out.print("Enter the manufacturer: ");
     testMan = keyboard.nextLine();

     // Get the model number.
     System.out.print("Enter the model number: ");
```

```
        testMod = keyboard.nextLine();

        // Get the retail price.
        System.out.print("Enter the retail price: ");
        testPrice = keyboard.nextDouble();

        // Create an instance of the CellPhone class,
        // passing the data that was entered as arguments
        // to the constructor.
        CellPhone phone = new CellPhone(testMan, testMod, testPrice);

        // Get the data from the phone and display it.
        System.out.println();
        System.out.println("Here is the data that you provided:");
        System.out.println("Manufacturer: " + phone.getManufact());
        System.out.println("Model number: " + phone.getModel());
        System.out.println("Retail price: " + phone.getRetailPrice());
    }
}
```

**Program Output with Example Input Shown in Bold**

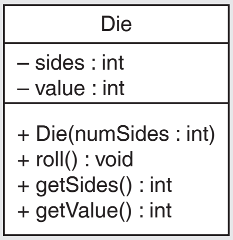
```
Enter the manufacturer: Acme Electronics [Enter]
Enter the model number: M1000 [Enter]
Enter the retail price: 199.99 [Enter]

Here is the data that you provided:
Manufacturer: Acme Electronics
Model number: M1000
Retail price: $199.99
```

## In the Spotlight:
## Simulating Dice with Objects

Dice traditionally have six sides, representing the values 1 through 6. Some games, however, use specialized dice that have a different number of sides. For example, the fantasy role-playing game *Dungeons and Dragons*® uses dice with four, six, eight, ten, twelve, and twenty sides.

Suppose you are writing a program that needs to roll simulated dice with various numbers of sides. A simple approach would be to write a `Die` class with a constructor that accepts the number of sides as an argument. The class would also have appropriate methods for rolling the die, and getting the die's value. Figure 6-24 shows the UML diagram for such a class, and Code Listing 6-14 shows the code.

**Figure 6-24** UML diagram for the `Die` class


Die
– sides : int
– value : int
+ Die(numSides : int)
+ roll() : void
+ getSides() : int
+ getValue() : int

**Code Listing 6-14** `(Die.java)`

```java
import java.util.Random;

/**
   The Die class simulates a six-sided die.
*/

public class Die
{
   private int sides;   // Number of sides
   private int value;   // The die's value

   /**
      The constructor performs an initial
      roll of the die.
      @param numSides The number of sides for this die.
   */

   public Die(int numSides)
   {
      sides = numSides;
      roll();
   }

   /**
      The roll method simulates the rolling of
      the die.
   */

   public void roll()
   {
      // Create a Random object.
      Random rand = new Random();

      // Get a random value for the die.
      value = rand.nextInt(sides) + 1;
   }

38    /**
39       getSides method
40       @return The number of sides for this die.
41    */
42 
43    public int getSides()
44    {
45       return sides;
46    }
47 
48    /**
49       getValue method
50       @return The value of the die.
51    */
52 
53    public int getValue()
54    {
55       return value;
56    }
57 }
```

Let's take a closer look at the code for the class:

Lines 9 and 10: These statements declare two `int` fields. The `sides` field will hold the number of sides that the die has, and the `value` field will hold the value of the die once it has been rolled.

Lines 18–22: This is the constructor. Notice that the constructor has a parameter for the number of sides. The parameter is assigned to the `sides` field in line 20. Line 21 calls the `roll` method, which simulates the rolling of the die.

Lines 29–36: This is the `roll` method, which simulates the rolling of the die. In line 32 a `Random` object is created, and it is referenced by the `rand` variable. Line 35 uses the `Random` object to get a random number that is in the appropriate range for this particular die. For example, if the `sides` field is set to 6, the expression `rand.nextInt(sides) + 1` will return a random integer in the range of 1 through 6. The random number is assigned to the `value` field.

Lines 43–46: This is the `getSides` method, an accessor that returns the `sides` field.

Lines 53–56: This is the `getValue` method, an accessor that returns the `value` field.

The program in Code Listing 6-15 demonstrates the class. It creates two instances of the Die class: one with six sides, and the other with twelve sides. It then simulates five rolls of the dice.

**Code Listing 6-15** (`DiceDemo.java`)

```
/**
   This program simulates the rolling of dice.
*/

```

```
public class DiceDemo
{
   public static void main(String[] args)
   {
      final int DIE1_SIDES = 6;   // Number of sides for die #1
      final int DIE2_SIDES = 12;  // Number of sides for die #2
      final int MAX_ROLLS = 5;    // Number of times to roll

      // Create two instances of the Die class.
      Die die1 = new Die(DIE1_SIDES);
      Die die2 = new Die(DIE2_SIDES);

      // Display the initial state of the dice.
      System.out.println("This simulates the rolling of a " +
                         DIE1_SIDES + " sided die and a " +
                         DIE2_SIDES + " sided die.");

      System.out.println("Initial value of the dice:");
      System.out.println(die1.getValue() + " " + die2.getValue());

      // Roll the dice five times.
      System.out.println("Rolling the dice " + MAX_ROLLS + " times.");

      for (int i = 0; i < MAX_ROLLS; i++)
      {
         // Roll the dice.
         die1.roll();
         die2.roll();

         // Display the values of the dice.
         System.out.println(die1.getValue() + " " + die2.getValue());
      }
   }
}
```

**Program Output**

```
This simulates the rolling of a 6 sided die and a 12 sided die.
Initial value of the dice:
2 7
Rolling the dice 5 times.
3 5
5 2
2 1
4 1
5 9
```

Let’s take a closer look at the program:

| | |
|---|---|
| Lines 9–11: | These statements declare three constants. `DIE1_SIDES` is the number of sides for the first die (6), `DIE2_SIDES` is the number of sides for the second die (12), and `MAX_ROLLS` is the number of times to roll the die (5). |
| Lines 14–15: | These statements create two instances of the `Die` class. Notice that `DIE1_SIDES`, which is 6, is passed to the constructor in line 14, and `DIE2_SIDES`, which is 12, is passed to the constructor in line 15. As a result, `die1` will reference a `Die` object with six sides, and `die2` will reference a `Die` object with twelve sides. |
| Lines 23: | This statement displays the initial value of both `Die` objects. (Recall that the `Die` class constructor performs an initial roll of the die.) |
| Lines 28–36: | This `for` loop iterates five times. Each time the loop iterates, line 31 calls the `die1` object’s `roll` method, and line 32 calls the `die2` object’s `roll` method. Line 35 displays the value of both dice. |

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

6.18 How is a constructor named?

6.19 What is a constructor’s return type?

6.20 Assume that the following is a constructor, which appears in a class:

```
ClassAct(int number)
{
     item = number;
}
```

a) What is the name of the class that this constructor appears in?

b) Write a statement that creates an object from the class and passes the value 25 as an argument to the constructor.

## 6.5 Passing Objects as Arguments

**CONCEPT: When an object is passed as an argument to a method, the object’s address is passed into the method’s parameter variable. As a result, the parameter references the object.**

When you are developing applications that work with objects, you will often need to write methods that accept objects as arguments. For example, suppose that a program is using the `Die` class that was previously shown in Code Listing 6-14. The following code shows a method named `showDieSides` that accepts a `Die` object as an argument:

```
void showDieSides(Die d)
{
    System.out.println("This die has " + d.getSides() +
                       " sides.");
}
```

The following code sample shows how we might create a `Die` object, and then pass it as an argument to the `showDieSides` method:

```
Die myDie = new Die(6);
showDieSides(myDie)
```

When you pass an object as an argument, the thing that is passed into the parameter variable is the object's memory address. As a result, the parameter variable references the object, and the method has access to the object.

The program shown in Code Listing 6-16 gives a complete demonstration. It creates two `Die` objects: one with six sides, and the other with twenty sides. It passes each object to a method named `rollDie` that displays the die's sides, rolls the die, and displays the die's value.

**Code Listing 6-16** (`DieArgument.java`)

```
/**
   This program rolls a 6-sided die and
   a 20-sided die.
*/

public class DieArgument
{
   public static void main(String[] args)
   {
      final int SIX_SIDES = 6;
      final int TWENTY_SIDES = 20;

      // Create a 6-sided die.
      Die sixDie = new Die(SIX_SIDES);

      // Create a 20-sided die.
      Die twentyDie = new Die(TWENTY_SIDES);

      // Roll the dice.
      rollDie(sixDie);
      rollDie(twentyDie);
   }

   /**
      This method simulates a die roll, displaying
      the die's number of sides and value.
      @param d The Die object to roll.
   */

   public static void rollDie(Die d)
   {
      // Display the number of sides.
      System.out.println("Rolling a " + d.getSides() +
                         " sided die.");

      // Roll the die.
      d.roll();

      // Display the die's value.
      System.out.println("The die's value: " + d.getValue());
   }
}
```

**Program Output**

```
Rolling a 6 sided die.
The die's value: 3
Rolling a 20 sided die.
The die's value: 19
```

## In the Spotlight:

## Simulating the Game of Cho-Han

Cho-Han is a traditional Japanese gambling game in which a dealer uses a cup to roll two six-sided dice. The cup is placed upside down on a table so that the value of the dice is concealed. Players then wager on whether the sum of the dice values is even (Cho) or odd (Han). The winner or winners take all of the wagers, or the house takes them if there are no winners.

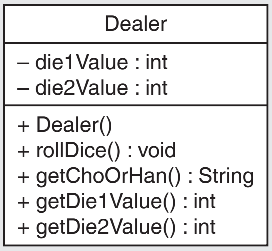
We will develop a program that simulates a simplified variation of the game. The simulated game will have a dealer and two players. The players will not wager money, but will simply guess whether the sum of the dice values is even (Cho) or odd (Han). One point will be awarded to the player, or players, correctly guessing the outcome. The game will play for five rounds, and the player with the most points is the grand winner.

In the program, we will use the `Die` class that was introduced in Code Listing 6-14. We will create two instances of the class to represent two six-sided dice. In addition to the `Die` class, we will write the following classes:

- `Dealer` class: We will create an instance of this class to represent the dealer. It will have the ability to roll the dice, report the value of the dice, and report whether the total dice value is Cho or Han.
- `Player` class: We will create two instances of this class to represent the players. Instances of the `Player` class can store the player's name, make a guess between Cho and Han, and be awarded points.

First, let's look at the `Dealer` class. Figure 6-25 shows a UML diagram for the class, and Code Listing 6-17 shows the code.

**Figure 6-25** UML diagram for the `Dealer` class


Dealer
– die1Value : int
– die2Value : int
+ Dealer()
+ rollDice() : void
+ getChoOrHan() : String
+ getDie1Value() : int
+ getDie2Value() : int

**Code Listing 6-17** (`Dealer.java`)

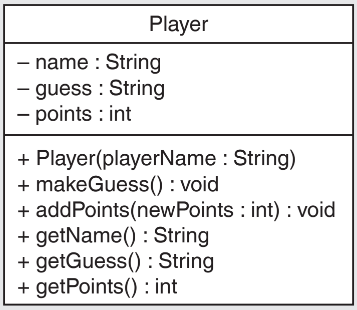
```java
/**
   Dealer class for the game of Cho-Han
*/

public class Dealer
{
   private int die1Value;  // The value of die #1
   private int die2Value;  // The value of die #2

   /**
      Constructor
   */

   public Dealer()
   {
      die1Value = 0;
      die2Value = 0;
   }

   /**
      The rollDice method rolls the dice and saves
      their values.
   */

   public void rollDice()
   {
      final int SIDES = 6; // Number of sides for the dice

      // Create the two dice. (This also rolls them.)
      Die die1 = new Die(SIDES);
      Die die2 = new Die(SIDES);

      // Record their values.
      die1Value = die1.getValue();
      die2Value = die2.getValue();
   }

   /**
      The getChoOrHan method returns the result of
      the dice roll, Cho or Han.
      @return Either "Cho (even)" or "Han (odd)"
   */

   public String getChoOrHan()
   {
      String result; // To hold the result

      // Get the sum of the dice.
      int sum = die1Value + die2Value;

      // Determine even or odd.
      if (sum % 2 == 0)
         result = "Cho (even)";
      else
         result = "Han (odd)";

      // Return the result.
      return result;
   }

   /**
      The getDie1Value method returns the value of
      die #1.
      @return The die1Value field
   */

   public int getDie1Value()
   {
      return die1Value;
   }

   /**
      The getDie2Value method returns the value of
      die #2.
      @return The die2Value field
   */

   public int getDie2Value()
   {
      return die2Value;
   }
}
```

Let's take a closer look at the code for the `Dealer` class:

- Lines 7 and 8 declare the fields `die1Value` and `die2Value`. These fields will hold the value of the two dice after they have been rolled.
- The constructor, in lines 14 through 18, initializes the `die1Value` and `die2Value` fields to 0.
- The `rollDice` method, in lines 25 through 36, simulates the rolling of the dice. Lines 30 and 31 create two `Die` objects. Recall that the `Die` class constructor performs an initial roll of the die, so there is no need to call the `Die` objects' `roll` method. Lines 34 and 35 save the value of the dice in the `die1Value` and `die2Value` fields.
- The `getChoOrHan` method, in lines 44 through 59, returns a string indicating whether the sum of the dice is Cho (even) or Han (odd).
- The `getDie1Value` method, in lines 67 through 70, returns the value of the first die (stored in the `die1Value` field).
- The `getDie2Value` method, in lines 78 through 81, returns the value of the second die (stored in the `die2Value` field).

Now let's look at the `Player` class. Figure 6-26 shows a UML diagram for the class, and Code Listing 6-18 shows the code.

**Figure 6-26** UML diagram for the `Player` class


Player
– name : String
– guess : String
– points : int
+ Player(playerName : String)
+ makeGuess() : void
+ addPoints(newPoints : int) : void
+ getName() : String
+ getGuess() : String
+ getPoints() : int

**Code Listing 6-18** `(Player.java)`

```
import java.util.Random;

/**
   Player class for the game of Cho-Han
*/

public class Player
{
   private String name;     // The player's name
   private String guess;    // The player's guess
   private int points;      // The player's points

   /**
      Constructor
      @param playerName The player's name.
   */

   public Player(String playerName)
   {
      name = playerName;
      guess = "";
      points = 0;
   }

   /**
      The makeGuess method causes the player to guess
      either "Cho (even)" or "Han (odd)".
   */

   public void makeGuess()
   {
      // Create a Random object.
      Random rand = new Random();

      // Get a random number, either 0 or 1.
      int guessNumber = rand.nextInt(2);

      // Convert the random number to a guess of
      // either "Cho (even)" or "Han (odd)".
      if (guessNumber == 0)
         guess = "Cho (even)";
      else
         guess = "Han (odd)";
   }

   /**
      The addPoints method adds a specified number of
      points to the player's current balance.
      @newPoints The points to add.
   */

   public void addPoints(int newPoints)
   {
      points += newPoints;
   }

   /**
      The getName method returns the player's name.
      @return The value of the name field.
   */

   public String getName()
```

```
63    {
64       return name;
65    }
66
67    /**
68       The getGuess method returns the player's guess.
69       @return The value of the guess field.
70    */
71
72    public String getGuess()
73    {
74       return guess;
75    }
76
77    /**
78       The getPoints method returns the player's points
79       @return The value of the points field.
80    */
81
82    public int getPoints()
83    {
84       return points;
85    }
86 }
```

Here's a summary of the code for the `Player` class:

- Lines 9 through 11 declare the fields `name`, `guess`, and `points`. These fields will hold the player's name, the player's guess, and the number of points the player has earned.
- The constructor, in lines 18 through 23, accepts an argument for the player's name, which is assigned to the `name` field. The `guess` field is assigned an empty string, and the `points` field is set to 0.
- The `makeGuess` method, in lines 30 through 44, causes the player to make a guess. The method generates a random number that is either a 0 or a 1. The `if` statement that begins at line 40 assigns the string "Cho (even)" to the `guess` field if the random number is 0, or it assigns the string "Han (odd)" to the `guess` field if the random number is 1.
- The `addPoints` method, in lines 52 through 55, adds the number of points specified by the argument to the player's `point` field.
- The `getName` method, in lines 62 through 65, returns the player's name.
- The `getGuess` method, in lines 72 through 75, returns the player's guess.
- The `getPoints` method, in lines 82 through 85, returns the player's points.

Code Listing 6-19 shows the program that uses these classes to simulate the game. The `main` method simulates five rounds of the game, displaying the results of each round, and then displays the overall game results.

**Code Listing 6-19** (`ChoHan.java`)

```java
import java.util.Scanner;

public class ChoHan
{
   public static void main(String[] args)
   {
      final int MAX_ROUNDS = 5;  // Number of rounds
      String player1Name;        // First player's name
      String player2Name;        // Second player's name

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the players' names.
      System.out.print("Enter the first player's name: ");
      player1Name = keyboard.nextLine();
      System.out.print("Enter the second player's name: ");
      player2Name = keyboard.nextLine();

      // Create the dealer.
      Dealer dealer = new Dealer();

      // Create the two players.
      Player player1 = new Player(player1Name);
      Player player2 = new Player(player2Name);

      // Play the rounds.
      for (int round = 0; round < MAX_ROUNDS; round++)
      {
         System.out.println("----------------------------");
         System.out.printf("Now playing round %d.\n", round + 1);

         // Roll the dice.
         dealer.rollDice();

         // The players make their guesses.
         player1.makeGuess();
         player2.makeGuess();

         // Determine the winner of this round.
         roundResults(dealer, player1, player2);
      }

      // Display the grand winner.
      displayGrandWinner(player1, player2);
   }

   /**
      The roundResults method determines the results of
      the current round.
      @param dealer The Dealer object
      @param player1 Player #1 object
      @param player2 Player #2 object
   */

   public static void roundResults(Dealer dealer, Player player1,
                                   Player player2)
   {
      // Show the dice values.
      System.out.printf("The dealer rolled %d and %d.\n",
                        dealer.getDie1Value(), dealer.getDie2Value());
      System.out.printf("Result: %s\n", dealer.getChoOrHan());

      // Check each player's guess and award points.
      checkGuess(player1, dealer);
      checkGuess(player2, dealer);
   }

   /**
      The checkGuess method checks a player's guess against
      the dealer's result.
      @param player The Player object to check.
      @param dealer The Dealer object.
   */

   public static void checkGuess(Player player, Dealer dealer)
   {
      final int POINTS_TO_ADD = 1; // Points to award winner
      String guess = player.getGuess();           // Player's guess
      String choHanResult = dealer.getChoOrHan(); // Cho or Han

      // Display the player's guess.
      System.out.printf("The player %s guessed %s.\n",
                        player.getName(), player.getGuess());

      // Award points if the player guessed correctly.
      if (guess.equalsIgnoreCase(choHanResult))
      {
         player.addPoints(POINTS_TO_ADD);
         System.out.printf("Awarding %d point(s) to %s.\n",
                           POINTS_TO_ADD, player.getName());
      }
   }

   /**
```

```
      The displayGrandWinner method displays the game's grand winner.
      @param player1 Player #1
      @param player2 Player #2
   */

   public static void displayGrandWinner(Player player1, Player player2)
   {
      System.out.println("----------------------------");
      System.out.println("Game over. Here are the results:");
      System.out.printf("%s: %d points.\n", player1.getName(),
                        player1.getPoints());
      System.out.printf("%s: %d points.\n", player2.getName(),
                        player2.getPoints());

      if (player1.getPoints() > player2.getPoints())
         System.out.println(player1.getName() + " is the grand winner!");
      else if (player2.getPoints() > player1.getPoints())
         System.out.println(player2.getName() + " is the grand winner!");
      else
         System.out.println("Both players are tied!");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the first player's name: Chelsea [Enter]
Enter the second player's name: Chris [Enter]
----------------------------
Now playing round 1.
The dealer rolled 3 and 6.
Result: Han (odd)
The player Chelsea guessed Han (odd).
Awarding 1 point(s) to Chelsea.
The player Chris guessed Han (odd).
Awarding 1 point(s) to Chris.
----------------------------
Now playing round 2.
The dealer rolled 4 and 5.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Cho (even).
----------------------------
Now playing round 3.
The dealer rolled 5 and 6.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Han (odd).
Awarding 1 point(s) to Chris.
```

```
----------------------------
Now playing round 4.
The dealer rolled 1 and 6.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Cho (even).
----------------------------
Now playing round 5.
The dealer rolled 6 and 6.
Result: Cho (even)
The player Chelsea guessed Han (odd).
The player Chris guessed Cho (even).
Awarding 1 point(s) to Chris.
----------------------------
Game over. Here are the results:
Chelsea: 1 points.
Chris: 3 points.
Chris is the grand winner!
```

Let's look at the code. Here is a summary of the `main` method:

- Lines 7 through 9 make the following declarations: `MAX_ROUNDS`—the number of rounds to play, `player1Name`—to hold the name of player #1, and `player2Name`—to hold the name of player #2.
- Lines 15 through 18 prompt the user to enter the players' names.
- Line 21 creates an instance of the `Dealer` class. The object represents the dealer, and is referenced by the `dealer` variable.
- Line 24 creates an instance of the `Player` class. The object represents player #1, and is referenced by the `player1` variable. Notice that `player1Name` is passed as an argument to the constructor.
- Line 25 creates another instance of the `Player` class. The object represents player #2, and is referenced by the `player2` variable. Notice that `player2Name` is passed as an argument to the constructor.
- The `for` loop that begins in line 28 iterates five times, causing the simulation of five rounds of the game. The loop performs the following actions:
  - Line 34 causes the dealer to roll the dice.
  - Line 37 causes player #1 to make a guess (Cho or Han).
  - Line 38 causes player #2 to make a guess (Cho or Han).
  - Line 41 passes the `dealer`, `player1`, and `player2` objects to the `roundResults` method. The method displays the results of this round.
- Line 45 passes the `player1` and `player2` objects to the `displayGrandWinner` method, which displays the grand winner of the game.

The `roundResults` method, which displays the results of a round, appears in lines 56 through 67. Here is a summary of the method:

- The method accepts references to the `dealer`, `player1`, and `player2` objects as arguments.
- The statement in lines 60 and 61 displays the value of the two dice.

- Line 62 calls the `dealer` object's `getChoOrHan` method to display the results, Cho or Han.
- Line 65 calls the `checkGuess` method, passing the `player1` and `dealer` objects as arguments. The `checkGuess` method compares a player's guess to the dealer's result (Cho or Han), and awards points to the player, if the guess is correct.
- Line 66 calls the `checkGuess` method, passing the `player2` and `dealer` objects as arguments.

The `checkGuess` method, which compares a player's guess to the dealer's result, awarding points to the player for a correct guess, appears in lines 76 through 93. Here is a summary of the method:

- The method accepts references to a `Player` object and the `Dealer` object as arguments.
- Line 78 declares the constant `POINTS_TO_ADD`, set to the value 1, which is the number of points to add to the player's balance if the player's guess is correct.
- Line 79 assigns the player's guess to the `String` object `guess`.
- Line 80 assigns the dealer's results (Cho or Han) to the `String` object `choHanResult`.
- The statement in lines 83 and 84 displays the player's name and guess.
- The `if` statement in line 87 compares the player's guess to the dealer's result. If they match, then the player guessed correctly, and line 89 awards points to the player.

The `displayGrandWinner` method, which displays the grand winner of the game, appears in lines 101 through 116. Here is a summary of the method:

- The method accepts references to the `player1` and `player2` objects.
- The statements in lines 105 through 108 display both players' names and points.
- The `if-else-if` statement that begins in line 110 determines which of the two players has the highest score, and displays that player's name as the grand winner. If both players have the same score, a tie is declared.

## 6.6 Overloading Methods and Constructors

**CONCEPT: Two or more methods in a class may have the same name as long as their parameter lists are different. This also applies to constructors.**

Method overloading is an important part of object-oriented programming. When a method is *overloaded*, it means that multiple methods in the same class have the same name, but use different types of parameters. Method overloading is important because sometimes you need several different ways to perform the same operation. For example, suppose a class has the following two methods:

```
public int add(int num1, int num2)
{
   int sum = num1 + num2;
   return sum;
}
```

```
public String add(String str1, String str2)
{
   String combined = str1 + str2;
   return combined;
}
```

Both of these methods are named `add`. They both take two arguments, which are added together. The first one accepts two `int` arguments and returns their sum. The second accepts two `String` references and returns a reference to a `String` that is a concatenation of the two arguments. When we write a call to the `add` method, the compiler must determine which one of the overloaded methods we intended to call.

The process of matching a method call with the correct method is known as *binding*. When an overloaded method is being called, Java uses the method's name and parameter list to determine which method to bind the call to. If two `int` arguments are passed to the `add` method, the version of the method with two `int` parameters is called. Likewise, when two `String` arguments are passed to `add`, the version with two `String` parameters is called.

Java uses a method's signature to distinguish it from other methods of the same name. A method's *signature* consists of the method's name and the data types of the method's parameters, in the order that they appear. For example, here are the signatures of the `add` methods that were previously shown:

```
add(int, int)
add(String, String)
```

Note that the method's return type is *not* part of the signature. For this reason, the following `add` method cannot be added to the same class with the previous ones:

```
public int add(String str1, String str2)
{
   int sum = Integer.parstInt(str1) + Integer.parseInt(str2);
   return sum;
}
```

Because the return type is not part of the signature, this method's signature is the same as that of the other `add` method that takes two `String` arguments. For this reason, an error message will be issued when a class containing all of these methods is compiled.

Constructors can also be overloaded, which means that a class can have more than one constructor. The rules for overloading constructors are the same for overloading other methods: Each version of the constructor must have a different parameter list. As long as each constructor has a unique signature, the compiler can tell them apart. For example, the `Rectangle` class that we discussed earlier could have the following two constructors:

```
public Rectangle()
{
   length = 0.0;
   width = 0.0;
}
```

```
public Rectangle(double len, double w)
{
   length = len;
   width = w;
}
```

The first constructor shown here accepts no arguments, and assigns 0.0 to the `length` and `width` fields. The second constructor accepts two arguments, which are assigned to the `length` and `width` fields. The following code shows an example of how each constructor is called:

```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

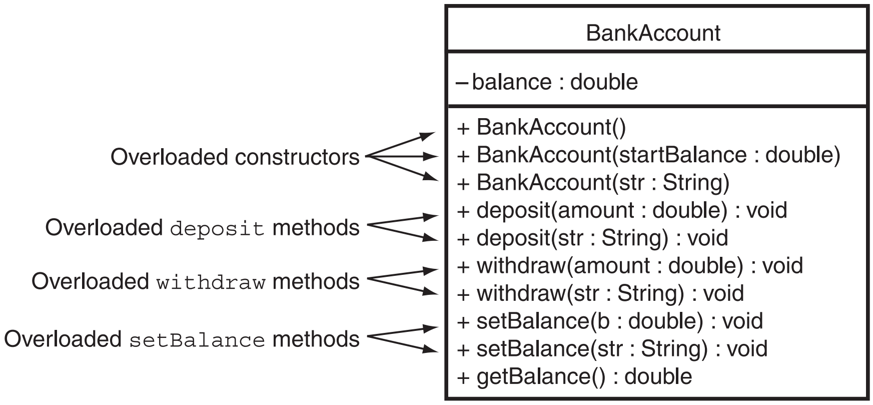
The first statement creates a `Rectangle` object, referenced by the `box1` variable, and executes the no-arg constructor. Its `length` and `width` fields will be set to 0.0. The second statement creates another `Rectangle` object, referenced by the `box2` variable, and executes the second constructor. Its `length` and `width` fields will be set to 5.0 and 10.0, respectively.

Recall that Java provides a default constructor only when you do not write any constructors for a class. If a class has a constructor that accepts arguments, but it does not have a no-arg constructor, you cannot create an instance of the class without passing arguments to the constructor. Therefore, any time you write a constructor for a class, and that constructor accepts arguments, you should also write a no-arg constructor if you want to be able to create instances of the class without passing arguments to the constructor.

## The `BankAccount` Class

Now we will look at the `BankAccount` class. Objects that are created from this class will simulate bank accounts, allowing us to have a starting balance, make deposits, make withdrawals, and get the current balance. A UML diagram for the `BankAccount` class is shown in Figure 6-27. In the figure, the overloaded constructors and overloaded methods are pointed out. Note that the extra annotation is not part of the UML diagram. It is there to draw attention to the items that are overloaded.

**Figure 6-27** UML diagram for the BankAccount class


BankAccount
– balance : double
+ BankAccount()
+ BankAccount(startBalance : double)
+ BankAccount(str : String)
+ deposit(amount : double) : void
+ deposit(str : String) : void
+ withdraw(amount : double) : void
+ withdraw(str : String) : void
+ setBalance(b : double) : void
+ setBalance(str : String) : void
+ getBalance() : double
Overloaded constructors
Overloaded deposit methods
Overloaded withdraw methods
Overloaded setBalance methods

As you can see from the diagram, the class has three overloaded constructors. Also, the class has two overloaded methods named `deposit`, two overloaded methods named `withdraw`, and two overloaded methods named `setBalance`. The last method, `getBalance`, is not overloaded. Code Listing 6-20 shows the code for the class.

**Code Listing 6-20** `(BankAccount.java)`

```
/**
   The BankAccount class simulates a bank account.
*/

public class BankAccount
{
   private double balance;      // Account balance

   /**
      This constructor sets the starting balance
      at 0.0.
   */

   public BankAccount()
   {
      balance = 0.0;
   }

   /**
      This constructor sets the starting balance
      to the value passed as an argument.
      @param startBalance The starting balance.
   */

   public BankAccount(double startBalance)
   {
      balance = startBalance;
   }

   /**
      This constructor sets the starting balance
      to the value in the String argument.
      @param str The starting balance, as a String.
   */

   public BankAccount(String str)
   {
      balance = Double.parseDouble(str);
   }

   /**
      The deposit method makes a deposit into
      the account.
      @param amount The amount to add to the
                    balance field.
   */

   public void deposit(double amount)
   {
      balance += amount;
   }

   /**
      The deposit method makes a deposit into
      the account.
      @param str The amount to add to the
                 balance field, as a String.
   */

   public void deposit(String str)
   {
      balance += Double.parseDouble(str);
   }

   /**
      The withdraw method withdraws an amount
      from the account.
      @param amount The amount to subtract from
                    the balance field.
   */

   public void withdraw(double amount)
   {
      balance -= amount;
   }

   /**
      The withdraw method withdraws an amount
      from the account.
      @param str The amount to subtract from
                 the balance field, as a String.
   */

   public void withdraw(String str)
   {
      balance -= Double.parseDouble(str);
   }

   /**
      The setBalance method sets the account balance.
      @param b The value to store in the balance field.
   */

   public void setBalance(double b)
   {
      balance = b;
   }

   /**
      The setBalance method sets the account balance.
      @param str The value, as a String, to store in
                 the balance field.
   */

   public void setBalance(String str)
   {
      balance = Double.parseDouble(str);
   }

   /**
      The getBalance method returns the
      account balance.
      @return The value in the balance field.
   */

   public double getBalance()
   {
      return balance;
   }
}
```

The class has one field, `balance`, which is a `double`. This field holds an account's current balance. Here is a summary of the class's overloaded constructors:

- The first constructor is a no-arg constructor. It sets the `balance` field to 0.0. If we wish to execute this constructor when we create an instance of the class, we simply pass no constructor arguments. Here is an example:

  ```
  BankAccount account = new BankAccount();
  ```

- The second constructor has a `double` parameter variable, `startBalance`, which is assigned to the `balance` field. If we wish to execute this constructor when we create an instance of the class, we pass a `double` value as a constructor argument. Here is an example:

  ```
  BankAccount account = new BankAccount(1000.0);
  ```

- The third constructor has a `String` parameter variable, `str`. It is assumed that the `String` contains a string representation of the account's balance. The method uses the `Double.parseDouble` method to convert the string to a `double`, and then assigns it to the `balance` field. If we wish to execute this constructor when we create an instance of the class, we pass a reference to a `String` as a constructor argument. Here is an example:

  ```
  BankAccount account = new BankAccount("1000.0");
  ```

  This constructor is provided as a convenience. If the class is used in a program that reads the account balance from a dialog box, or from a text file, the amount does not have to be converted from a string before it is passed to the constructor.

Here is a summary of the overloaded `deposit` methods:

- The first `deposit` method has a parameter, `amount`, which is a `double`. When the method is called, an amount that is to be deposited into the account is passed into this parameter. The value of the parameter is then added to value in the `balance` field.
- The second `deposit` method has a parameter, `str`, which is a reference to a `String`. It is assumed that the `String` contains a string representation of the amount to be deposited. The method uses the `Double.parseDouble` method to convert the string to a `double`, and then adds it to the `balance` field. For example, if we call the method and pass "500.0" as the argument, it will add 500.0 to the balance field. As with the overloaded constructors, this method is provided as a convenience for programs that read the amount to be deposited from a dialog box or a text file.

Here is a summary of the overloaded `withdraw` methods:

- The first `withdraw` method has a parameter, `amount`, which is a `double`. When the method is called, an amount that is to be withdrawn from the account is passed into this parameter. The value of the parameter is then subtracted from the value in the `balance` field.
- The second `withdraw` method has a parameter, `str`, which is a reference to a `String`. It is assumed that the `String` contains a string representation of the amount to be withdrawn. This amount is converted to a `double`, and then subtracted from the `balance` field. As with the overloaded constructors and `deposit` methods, this method is provided as a convenience.

Here is a summary of the overloaded `setBalance` methods:

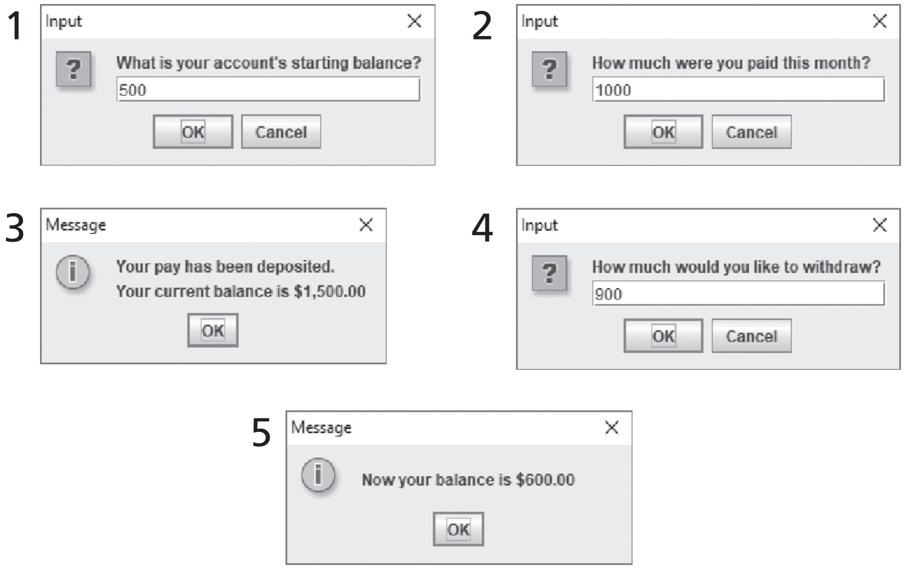
- The first `setBalance` method accepts a `double` argument, which is assigned to the `balance` field.
- The second `setBalance` method accepts a `String` reference as an argument. It is assumed that the `String` contains a string representation of the account's balance. The `String` is converted to a `double` and then assigned to the `balance` field. As with many of the other overloaded methods, this method is provided as a convenience.

The remaining method is `getBalance`. It returns the value in the `balance` field, which is the current account balance. The *AccountTest.java* program, shown in Code Listing 6-21, demonstrates the `BankAccount` class. Its output is shown in Figure 6-28.

**Code Listing 6-21** (AccountTest.java)

```
import javax.swing.JOptionPane; // For the JOptionPane class

/**
   This program demonstrates the BankAccount class.
*/

public class AccountTest
{
   public static void main(String[] args)
   {
      String input;    // To hold user input

      // Get the starting balance.
      input = JOptionPane.showInputDialog(
                 "What is your account's starting balance?");

      // Create a BankAccount object.
      BankAccount account = new BankAccount(input);

      // Get the amount of pay.
      input = JOptionPane.showInputDialog(
                 "How much were you paid this month?");

      // Deposit the user's pay into the account.
      account.deposit(input);

      // Display the new balance.
      JOptionPane.showMessageDialog(null,
         String.format("Your pay has been deposited.\n" +
                       "Your current balance is $%,.2f",
                       account.getBalance()));

      // Withdraw some cash from the account.
      input = JOptionPane.showInputDialog(
                 "How much would you like to withdraw?");
      account.withdraw(input);

      // Display the new balance
      JOptionPane.showMessageDialog(null,
         String.format("Now your balance is $%,.2f",
                       account.getBalance()));

      System.exit(0);
   }
}
```

**Figure 6-28** Interaction with the `AccountTest` program (Oracle Corporate Counsel)



1. What is your account's starting balance?
500

2. How much were you paid this month?
1000

3. Your pay has been deposited.
Your current balance is $1,500.00

4. How much would you like to withdraw?
900

5. Now your balance is $600.00

## Overloaded Methods Make Classes More Useful

You might be wondering why all those overloaded methods appear in the `BankAccount` class, especially because many of them weren't used by the demonstration program in Code Listing 6-21. After all, wouldn't it be simpler if the class had only the methods we were going to use?

An object's purpose is to provide a specific service. The service provided by the `BankAccount` class is that it simulates a bank account. Any program that needs a simulated bank account can simply create a `BankAccount` object and then use its methods to put the simulation into action. Because the `BankAccount` class has numerous overloaded methods, it is much more flexible than it would be if it provided only one way to perform every operation. By providing overloaded constructors, `deposit` methods, `withdraw` methods, and `setBalance` methods, we made the `BankAccount` class useful to programs other than our simple demonstration program. This is an important consideration to keep in mind when you design classes of your own.

## 6.7 Scope of Instance Fields

**CONCEPT: Instance fields are visible to all of the class's instance methods.**

Recall from Chapter 2 that a variable's scope is the part of a program where the variable may be accessed by its name. A variable's name is visible only to statements inside the variable's scope. The location of a variable's declaration determines the variable's scope.

In this chapter you have seen variables declared as instance fields in a class. An instance field can be accessed by any instance method in the same class as the field. If an instance field is declared with the `public` access specifier, it can also be accessed by code outside the class.

## Shadowing

In Chapter 2 you saw that you cannot have two local variables with the same name in the same scope. This applies to parameter variables as well. A parameter variable is, in essence, a local variable. So, you cannot give a parameter variable and a local variable in the same method the same name.

However, you can have a local variable or a parameter variable with the same name as a field. When you do, the name of the local or parameter variable *shadows* the name of the field. This means that the field name is hidden by the name of the local or parameter variable.

For example, assume that the `Rectangle` class's `setLength` method had been written in the following manner:

```
public void setLength(double len)
{
    int length;          // Local variable
    length = len;
}
```

In this code a local variable is given the same name as a field. Therefore, the local variable's name shadows the field's name. When the statement `length = len;` is executed, the value of `len` is assigned to the local variable `length`, not to the field. The unintentional shadowing of field names can cause elusive bugs, so you need to be careful not to give local variables the same names as fields.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

6.21 Is it required that overloaded methods have different return values, different parameter lists, or both?

6.22 What is a method's signature?

6.23 Look at the following class:

```
public class CheckPoint
{
   public void message(int x)
   {
      System.out.print("This is the first version ");
      System.out.println("of the method.");
   }
   public void message(String x)
   {
      System.out.print("This is the second version ");
      System.out.println("of the method.");
   }
}
```

What will the following code display?

```
CheckPoint cp = new CheckPoint();
cp.message("1");
cp.message(1);
```

6.24 How many default constructors may a class have?

## 6.8 Packages and `import` Statements

**CONCEPT: The classes in the Java API are organized into packages. An `import` statement tells the compiler which package a class is located in.**

In Chapter 2 you were introduced to the Java API, which is a standard library of prewritten classes. Each class in the Java API is designed for a specific purpose, and you can use the classes in your own programs. You've already used a few classes from the API, such as the `String` class, the `Scanner` class, the `JOptionPane` class, and the `Random` class.

All of the classes in the Java API are organized into packages. A *package* is simply a group of related classes. Each package also has a name. For example, the `Scanner` class is in the `java.util` package.

Many of the classes in the Java API are not automatically available to your program. Quite often, you have to *import* an API class in order to use it. You use the `import` key word to import a class. For example, the following statement is required to import the `Scanner` class:

```
import java.util.Scanner;
```

This statement tells the compiler that the `Scanner` class is located in the `java.util` package. Without this statement, the compiler will not be able to locate the `Scanner` class, and the program will not compile.

### Explicit and Wildcard `import` Statements

There are two types of `import` statements: explicit and wildcard. An *explicit* `import` statement identifies the package location of a single class. For example, the following statement explicitly identifies the location of the `Scanner` class:

```
import java.util.Scanner;
```

The `java.util` package contains several other classes as well as the `Scanner` class. For example, the `Random` class is also part of the `java.util` package. If a program needs to use the `Scanner` class and the `Random` class, it will have to import both of these classes. One way to do this is to write explicit `import` statements for each class, as follows:

```
import java.util.Scanner;
import java.util.Random;
```

Another way to import both of these classes is to use a wildcard `import` statement. A *wildcard import* statement tells the compiler to import all of the classes in a package. Here is an example:

```
import java.util.*;
```

The `.*` that follows the package name tells the compiler to import all the classes that are part of the `java.util` package. Using a wildcard `import` statement does not affect the performance or the size of your program. It merely tells the compiler that you want to make every class in a particular package available to your program.

## The `java.lang` Package

The Java API does have one package, `java.lang`, which is automatically imported into every Java program. This package contains general classes, such as `String` and `System`, that are fundamental to the Java programming language. You do not have to write an `import` statement for any class that is part of the `java.lang` package.

## Other API Packages

There are numerous packages in the Java API. Table 6-2 lists a few of them.

**Table 6-2** A few of the standard Java packages

| Package | Description |
|---|---|
| `java.io` | Provides classes that perform various types of input and output. |
| `java.lang` | Provides general classes for the Java language. This package is automatically imported. |
| `java.net` | Provides classes for network communications. |
| `java.security` | Provides classes that implement security features. |
| `java.sql` | Provides classes for accessing databases using structured query language. |
| `java.text` | Provides various classes for formatting text. |
| `java.util` | Provides various utility classes. |

See Appendix G, available on the book's companion Web site at www.pearsonhighered.com/gaddis, for a more detailed look at packages.

## 6.9 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities

So far you have learned the basics of writing a class, creating an object from the class, and using the object to perform operations. Although this knowledge is necessary to create an object-oriented application, it is not the first step. The first step is to analyze the problem that you are trying to solve and determine the classes that you will need. In this section, we will discuss a simple technique for finding the classes in a problem and determining their responsibilities.

### Finding the Classes

When developing an object-oriented application, one of your first tasks is to identify the classes that you will need to create. Typically, your goal is to identify the different types of real-world objects that are present in the problem, and then create classes for those types of objects within your application.

Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps:

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

#### Writing a Description of the Problem Domain

The *problem domain* is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are programming an application that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

> Joe's Automotive Shop services foreign cars, and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. Then the manager determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects such vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer, teacher, student, and so forth
- The results of a business event, such as a customer order, or in this case a service quote
- Recordkeeping items, such as customer histories and payroll records

### Identifying All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.) Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.

> Joe's Automotive Shop services foreign cars, and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. Then the manager determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

Notice that some of the nouns are repeated. The following list shows all of the nouns without duplication.

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
foreign cars
Joe's Automotive Shop
make
manager
Mercedes
model
name
Porsche
sales tax
service quote
shop
telephone number
total estimated charges
year

### Refining the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

#### 1. Some of the nouns really mean the same thing.

In this example, the following sets of nouns refer to the same thing:

- **car, cars,** and **foreign cars**
  These all refer to the general concept of a car.
- **Joe's Automotive Shop** and **shop**
  Both of these refer to the company "Joe's Automotive Shop."

We can settle on a single class for each of these. In this example we will arbitrarily eliminate **cars** and **foreign cars** from the list, and use the word **car**. Likewise, we will eliminate **Joe's Automotive Shop** from the list and use the word **shop**. The updated list of potential classes is as follows:

address
BMW
car
~~cars~~
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
manager
Mercedes
model
name
Porsche
sales tax
service quote
shop
telephone number
total estimated charges
year

Because **car, cars,** and **foreign cars** mean the same thing in this problem, we have eliminated **cars** and **foreign cars**. Also, because **Joe's Automotive Shop** and **shop** mean the same thing, we have eliminated **Joe's Automotive Shop**.

### 2. Some nouns might represent items that we do not need to be concerned with in order to solve the problem.

A quick review of the problem description reminds us of what our application should do: print a service quote. In this example we can eliminate two unnecessary classes from the list:

- We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the **manager** because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

At this point the updated list of potential classes is as follows:

address
BMW
car
~~cars~~
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
~~manager~~
Mercedes
model
name
Porsche
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

Our problem description does not direct us to process any information about the **shop**, or any information about the **manager**, so we have eliminated those from the list.

### 3. Some of the nouns might represent objects, not classes.

We can eliminate **Mercedes**, **Porsche**, and **BMW** as classes because, in this example, they all represent specific cars, and can be considered instances of a **car** class. In the description it refers to a specific car brought to the shop by a customer. Therefore, it would also represent an instance of a **cars** class. At this point the updated list of potential classes is as follows:

address
~~BMW~~
car
~~cars~~
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
~~manager~~
~~Mercedes~~
model
name
~~Porsche~~
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

We have eliminated **Mercedes**, **Porsche**, and **BMW** because they are all instances of a car class. That means that these nouns identify objects, not classes.

**TIP:** Some object-oriented designers take note of whether a noun is plural or singular. Sometimes a plural noun will indicate a class and a singular noun will indicate an object.

**4. Some of the nouns might represent simple values that can be stored in a primitive variable and do not require a class.**

Remember, a class contains fields and methods. Fields are related items that are stored within an object of the class, and define the object's state. Methods are actions or behaviors that may be performed by an object of the class. If a noun represents a type of item that would not have any identifiable fields or methods, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have fields and methods, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a primitive variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: **address**, **estimated labor charges**, **estimated parts charges**, **make**, **model**, **name**, **sales tax**, **telephone number**, **total estimated charges,** and **year**. These are all simple string or numeric values that can be stored in primitive variables. Here is the updated list of potential classes:

~~address~~
~~BMW~~
car
~~cars~~
customer
~~estimated labor charges~~
~~estimated parts charges~~
~~foreign cars~~
~~Joe's Automotive Shop~~
~~make~~
~~manager~~
~~Mercedes~~
~~model~~
~~name~~
~~Porsche~~
~~sales tax~~
service quote
~~shop~~
~~telephone number~~
~~total estimated charges~~
~~year~~

We have eliminated **address**, **estimated labor charges**, **estimated parts charges**, **make**, **model**, **name**, **sales tax**, **telephone number**, **total estimated charges**, and **year** as classes because they represent simple values that can be stored in primitive variables.

As you can see from the list, we have eliminated everything except **cars**, **customer**, and **service quote**. This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a `Car` class, a `Customer` class, and a `ServiceQuote` class.

## Identifying a Class's Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's *responsibilities* are as follows:

- The things that the class is responsible for knowing
- The actions that the class is responsible for doing

When you have identified the things that a class is responsible for knowing, you have identified the class's attributes. These values will be stored in fields. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its methods.

It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things that a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so brainstorming is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

### The `Customer` class

In the context of our problem domain, what must the `Customer` class know? The description directly mentions the following items, which are all attributes of a customer:

- The customer's name
- The customer's address
- The customer's telephone number

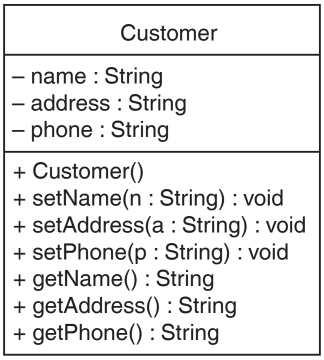
These are all values that can be represented as strings and stored in the class's fields. The `Customer` class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a `Customer` class might know the customer's email address. This particular problem domain does not mention that the customer's email address is used for any purpose, so we should not include it as a responsibility.

Now let's identify the class's methods. In the context of our problem domain, what must the `Customer` class do? The only obvious actions are as follows:

- Create an object of the `Customer` class
- Set and get the customer's name
- Set and get the customer's address
- Set and get the customer's telephone number

From this list we can see that the `Customer` class will have a constructor, as well as accessor and mutator methods for each of its fields. Figure 6-29 shows a UML diagram for the `Customer` class.

**Figure 6-29** UML diagram for the `Customer` class



### The `Car` Class

In the context of our problem domain, what must an object of the `Car` class know? The following items are all attributes of a car, and are mentioned in the problem domain:

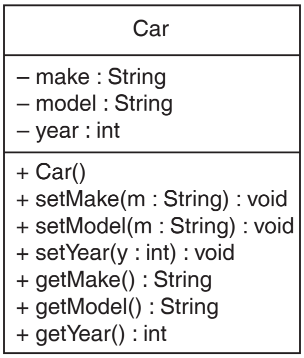
- The car's make
- The car's model
- The car's year

Now let's identify the class's methods. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of methods that we will find in most classes (constructors, accessors, and mutators). Specifically, the actions are:

- Create an object of the `Car` class
- Set and get the car's make
- Set and get the car's model
- Set and get the car's year

Figure 6-30 shows a UML diagram for the `Car` class at this point.

**Figure 6-30** UML diagram for the `Car` class



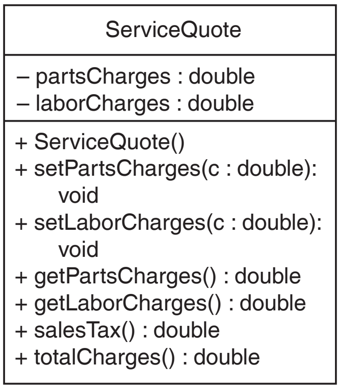
### The `ServiceQuote` Class

In the context of our problem domain, what must an object of the `ServiceQuote` class know? The problem domain mentions the following items:

- The estimated parts charges
- The estimated labor charges
- The sales tax
- The total estimated charges

Careful thought and a little brainstorming will reveal that two of these items are the results of calculations: sales tax and total estimated charges. These items are dependent on the values of the estimated parts and labor charges. In order to avoid the risk of holding stale data, we will not store these values in fields. Rather, we will provide methods that calculate these values and return them. The other methods that we will need for this class are a constructor and the accessors and mutators for the estimated parts charges and estimated labor charges fields. Figure 6-31 shows a UML diagram for the `ServiceQuote` class.

**Figure 6-31** UML diagram for the `ServiceQuote` class


ServiceQuote
– partsCharges : double
– laborCharges : double
+ ServiceQuote()
+ setPartsCharges(c : double): void
+ setLaborCharges(c : double): void
+ getPartsCharges() : double
+ getLaborCharges() : double
+ salesTax() : double
+ totalCharges() : double

## This Is Only the Beginning

You should consider the process that we have discussed in this section merely as a starting point. It's important to realize that designing an object-oriented application is an iterative process. It may take you several attempts to identify all of the classes that you will need, and determine all of their responsibilities. As the design process unfolds, you will gain a deeper understanding of the problem, and consequently you will see ways to improve the design.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

6.25 What is a problem domain?

6.26 When designing an object-oriented application, who should write a description of the problem domain?

6.27 How do you identify the potential classes in a problem domain description?

6.28 What are a class's responsibilities?

6.29 What two questions should you ask to determine a class's responsibilities?

6.30 Will all of a class's actions always be directly mentioned in the problem domain description?

See the `Amortization` Class Case Study, available on the book's companion Web site at www.pearsonhighered.com/gaddis, for an in-depth example using this chapter's topics.

## 6.10 Common Errors to Avoid

- **Putting a semicolon at the end of a method header.** A semicolon never appears at the end of a method header.
- **Declaring a variable to reference an object, but forgetting to use the `new` key word to create the object.** Declaring a variable to reference an object does not create an object. You must use the `new` key word to create the object.

- **Forgetting the parentheses that must appear after the class name, which appears after the `new` key word.** The name of a class appears after the `new` key word, and a set of parentheses appears after the class name. You must write the parentheses even if no arguments are passed to the constructor.
- **Forgetting to provide arguments when a constructor requires them**. When using a constructor that has parameter variables, you must provide arguments for them.
- **Trying to overload methods by giving them different return types.** Overloaded methods must have unique parameter lists.
- **Forgetting to write a no-arg constructor for a class that you want to be able to create instances of without passing arguments to the constructor.** If you write a constructor that accepts arguments, you must also write a no-arg constructor for the same class if you want to be able to create instances of the class without passing arguments to the constructor.
- **Unintentionally declaring a local variable with the same name as a field of the same class in a method.** When a method's local variable has the same name as a field in the same class, the local variable's name shadows the field's name.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This is a collection of programming statements that specify the fields and methods that a particular type of object may have.
   a. class
   b. method
   c. parameter
   d. instance
2. A class is analogous to a(n) __________.
   a. house
   b. blueprint
   c. drafting table
   d. architect
3. An object is a(n) __________.
   a. blueprint
   b. primitive data type
   c. variable
   d. instance of a class
4. This is a class member that holds data.
   a. method
   b. instance
   c. field
   d. constructor
5. This key word causes an object to be created in memory.
   a. `create`
   b. `new`
   c. `object`
   d. `construct`

6. This is a method that gets a value from a class's field, but does not change it.
   a. accessor
   b. constructor
   c. void
   d. mutator
7. This is a method that stores a value in a field or in some other way changes the value of a field.
   a. accessor
   b. constructor
   c. void
   d. mutator
8. When the value of an item is dependent on other data, and that item is not updated when the other data is changed, what has the value become?
   a. bitter
   b. stale
   c. asynchronous
   d. moldy
9. This is a method that is automatically called when an instance of a class is created.
   a. accessor
   b. constructor
   c. void
   d. mutator
10. When a local variable has the same name as a field, the local variable's name does this to the field's name.
    a. shadows
    b. complements
    c. deletes
    d. merges with
11. This is automatically provided for a class if you do not write one yourself.
    a. accessor method
    b. default instance
    c. default constructor
    d. variable declaration
12. Two or more methods in a class may have the same name, as long as this is different.
    a. their return values
    b. their access specifier
    c. their parameter lists
    d. their memory address
13. The process of matching a method call with the correct method is known as __________.
    a. matching
    b. binding
    c. linking
    d. connecting
14. A class's responsibilities are __________.
    a. the objects created from the class
    b. things the class knows
    c. actions the class performs
    d. both b and c

15. **True or False:** The `new` operator creates an instance of a class.
16. **True or False:** Each instance of a class has its own set of instance fields.
17. **True or False:** When you write a constructor for a class, it still has the default constructor that Java automatically provides.
18. **True or False:** A class may not have more than one constructor.
19. **True or False:** To find the classes needed for an object-oriented application, you identify all of the verbs in a description of the problem domain.

### Find the Error

1. Find the error in the following class:

```
public class MyClass
{
   private int x;
   private double y;
   public void MyClass(int a, double b)
   {
      x = a;
      y = b;
   }
}
```

2. Assume that the following method is a member of a class. Find the error.

```
public void total(int value1, value2, value3)
{
   return value1 + value2 + value3;
}
```

3. The following statement attempts to create a `Rectangle` object. Find the error.

```
Rectangle box = new Rectangle;
```

4. Find the error in the following class:

```
public class TwoValues
{
   private int x, y;
   public TwoValues()
   {
      x = 0;
   }
   public TwoValues()
   {
      x = 0;
      y = 0;
   }
}
```

5. Find the error in the following class:

```
public class FindTheError
{
   public int square(int number)
   {
      return number * number;
   }
   public double square(int number)
   {
      return number * number;
   }
}
```

## Algorithm Workbench

1. Design a class named `Pet`, which should have the following fields:
   - `name`. The `name` field holds the name of a pet.
   - `animal`. The `animal` field holds the type of animal that a pet is. Example values are “Dog”, “Cat”, and “Bird”.
   - `age`. The `age` field holds the pet’s age.

   The `Pet` class should also have the following methods:
   - `setName`. The `setName` method stores a value in the `name` field.
   - `setAnimal`. The `setAnimal` method stores a value in the `animal` field.
   - `setAge`. The `setAge` method stores a value in the `age` field.
   - `getName`. The `getName` method returns the value of the `name` field.
   - `getAnimal`. The `getAnimal` method returns the value of the `animal` field.
   - `getAge`. The `getAge` method returns the value of the `age` field.
     a. Draw a UML diagram of the class. Be sure to include notation showing each field and method’s access specification and data type. Also include notation showing any method parameters and their data types.
     b. Write the Java code for the `Pet` class.
2. Look at the following partial class definition, and then respond to the questions that follow it:

```
public class Book
{
   private String title;
   private String author;
   private String publisher;
   private int copiesSold;
}
```

   a. Write a constructor for this class. The constructor should accept an argument for each of the fields.
   b. Write accessor and mutator methods for each field.
   c. Draw a UML diagram for the class, including the methods you have written.

3. Consider the following class declaration:

```
public class Square
{
   private double sideLength;
   public double getArea()
   {
      return sideLength * sideLength;
   }
   public double getSideLength()
   {
      return sideLength;
   }
}
```

   a. Write a no-arg constructor for this class. It should assign the `sideLength` field the value 0.0.
   b. Write an overloaded constructor for this class. It should accept an argument that is copied into the `sideLength` field.

4. Look at the following description of a problem domain:

5. The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on the account. Each account has an interest rate.

6. Assume that you are writing an application that will calculate the amount of interest earned for a bank account.
   a. Identify the potential classes in this problem domain.
   b. Refine the list to include only the necessary class or classes for this problem.
   c. Identify the responsibilities of the class or classes.

## Short Answer

1. What is the difference between a class and an instance of a class?
2. A contractor uses a blueprint to build a set of identical houses. Are classes analogous to the blueprint or the houses?
3. What is an accessor method? What is a mutator method?
4. Is it a good idea to make fields private? Why or why not?
5. If a class has a private field, what has access to the field?
6. What is the purpose of the `new` key word?
7. Assume a program named *MailList.java* is stored in the *DataBase* folder on your hard drive. The program creates objects of the `Customer` and `Account` classes. Describe the steps that the compiler goes through in locating and compiling the `Customer` and `Account` classes.
8. Why are constructors useful for performing “start-up” operations?
9. Under what circumstances does Java automatically provide a default constructor for a class?

10. What do you call a constructor that accepts no arguments?
11. When the same name is used for two or more methods in the same class, how does Java tell them apart?
12. How does method overloading improve the usefulness of a class?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. `Employee` Class

Write a class named `Employee` that has the following fields:

- `name`. The `name` field references a `String` object that holds the employee's name.
- `idNumber`. The `idNumber` is an `int` variable that holds the employee's ID number.
- `department`. The `department` field references a `String` object that holds the name of the department where the employee works.
- `position`. The `position` field references a `String` object that holds the employee's job title.

The class should have the following constructors:

- A constructor that accepts the following values as arguments and assigns them to the appropriate fields: employee's name, employee's ID number, department, and position.
- A constructor that accepts the following values as arguments and assigns them to the appropriate fields: employee's name and ID number. The `department` and `position` fields should be assigned an empty string (`""`).
- A no-arg constructor that assigns empty strings (`""`) to the `name`, `department`, and `position` fields, and 0 to the `idNumber` field.

Write appropriate mutator methods that store values in these fields and accessor methods that return the values in these fields. Once you have written the class, write a separate program that creates three `Employee` objects to hold the following data:

| Name | ID Number | Department | Position |
|---|---|---|---|
| Susan Meyers | 47899 | Accounting | Vice President |
| Mark Jones | 39119 | IT | Programmer |
| Joy Rogers | 81774 | Manufacturing | Engineer |

The program should store this data in the three objects and then display the data for each employee on the screen.

### 2. `Car` Class

Write a class named `Car` that has the following fields:

- **`yearModel`.** The `yearModel` field is an `int` that holds the car's year model.
- **`make`.** The `make` field references a `String` object that holds the make of the car.
- **`speed`.** The `speed` field is an `int` that holds the car's current speed.

In addition, the class should have the following constructor and other methods.

- Constructor. The constructor should accept the car's year model and make as arguments. These values should be assigned to the object's `yearModel` and `make` fields. The constructor should also assign 0 to the `speed` field.
- Accessors. Appropriate accessor methods should get the values stored in an object's `yearModel`, `make`, and `speed` fields.
- `accelerate`. The `accelerate` method should add 5 to the `speed` field each time it is called.
- `brake`. The `brake` method should subtract 5 from the `speed` field each time it is called.

Demonstrate the class in a program that creates a `Car` object, and then calls the `accelerate` method five times. After each call to the `accelerate` method, get the current speed of the car and display it. Then call the `brake` method five times. After each call to the `brake` method, get the current speed of the car and display it.

### 3. Personal Information Class

VideoNote
The Personal Information Class Problem

Design a class that holds the following personal data: name, address, age, and phone number. Write appropriate accessor and mutator methods. Demonstrate the class by writing a program that creates three instances of it. One instance should hold your information, and the other two should hold your friends' or family members' information.

### 4. `RetailItem` Class

Write a class named `RetailItem` that holds data about an item in a retail store. The class should have the following fields:

- `description`. The `description` field references a `String` object that holds a brief description of the item.
- `unitsOnHand`. The `unitsOnHand` field is an `int` variable that holds the number of units currently in inventory.
- `price`. The `price` field is a `double` that holds the item's retail price.

Write a constructor that accepts arguments for each field, appropriate mutator methods that store values in these fields, and accessor methods that return the values in these fields. Once you have written the class, write a separate program that creates three `RetailItem` objects and stores the following data in them:

| | Description | Units on Hand | Price |
|---|---|---|---|
| Item #1 | Jacket | 12 | 59.95 |
| Item #2 | Designer Jeans | 40 | 34.95 |
| Item #3 | Shirt | 20 | 24.95 |

### 5. `Payroll` Class

Design a `Payroll` class that has fields for an employee's name, ID number, hourly pay rate, and number of hours worked. Write the appropriate accessor and mutator methods and a constructor that accepts the employee's name and ID number as arguments. The class should also have a method that returns the employee's gross pay, which is calculated as the number of hours worked multiplied by the hourly pay rate. Write a program that demonstrates the class by creating a `Payroll` object, then asking the user to enter the data for an employee. The program should display the amount of gross pay earned.

### 6. `TestScores` Class

Design a `TestScores` class that has fields to hold three test scores. The class should have a constructor, accessor and mutator methods for the test score fields, and a method that returns the average of the test scores. Demonstrate the class by writing a separate program that creates an instance of the class. The program should ask the user to enter three test scores, which are stored in the `TestScores` object. Then the program should display the average of the scores, as reported by the `TestScores` object.

### 7. `Circle` Class

Write a `Circle` class that has the following fields:

- **`radius`**: a `double`
- **`PI`**: a `final double` initialized with the value 3.14159

The class should have the following methods:

- **Constructor**. Accepts the radius of the circle as an argument.
- **Constructor**. A no-arg constructor that sets the `radius` field to 0.0.
- **`setRadius`**. A mutator method for the radius field.
- **`getRadius`**. An accessor method for the radius field.
- **`area`**. Returns the area of the circle, which is calculated as
  `area = PI * radius * radius`
- **`diameter`**. Returns the diameter of the circle, which is calculated as

  `diameter = radius * 2`
- **`circumference`**. Returns the circumference of the circle, which is calculated as

  `circumference = 2 * PI * radius`

Write a program that demonstrates the `Circle` class by asking the user for the circle's radius, creating a `Circle` object, and then reporting the circle's area, diameter, and circumference.

### 8. `Temperature` Class

Write a `Temperature` class that will hold a temperature in Fahrenheit, and provide methods to get the temperature in Fahrenheit, Celsius, and Kelvin. The class should have the following field:

- `fahrenheit` – A `double` that holds a Fahrenheit temperature.

The class should have the following methods:

- Constructor – The constructor accepts a Fahrenheit temperature (as a `double`) and stores it in the `ftemp` field.
- `setFahrenheit` – The `setFahrenheit` method accepts a Fahrenheit temperature (as a `double`) and stores it in the `ftemp` field.
- `getFahrenheit` – Returns the value of the `ftemp` field, as a Fahrenheit temperature (no conversion required).
- `celsius` – Returns the value of the `ftemp` field converted to Celsius.
- `kelvin` – Returns the value of the `ftemp` field converted to Kelvin.

Use the following formula to convert the Fahrenheit temperature to Celsius:

$$Celsius = (5/9) \times (Fahrenheit - 32)$$

Use the following formula to convert the Fahrenheit temperature to Kelvin:

$$Kelvin = ((5/9) \times (Fahrenheit - 32)) + 273$$

Demonstrate the `Temperature` class by writing a separate program that asks the user for a Fahrenheit temperature. The program should create an instance of the `Temperature` class, with the value entered by the user passed to the constructor. The program should then call the object's methods to display the temperature in Celsius and Kelvin.

### 9. Days in a Month

Write a class named `MonthDays`. The class's constructor should accept two arguments:

- An integer for the month (1 = January, 2 February, etc.).
- An integer for the year

The class should have a method named `numberOfDays` that returns the number of days in the specified month. The method should use the following criteria to identify leap years:

1. Determine whether the year is divisible by 100. If it is, then it is a leap year if and if only it is divisible by 400. For example, 2000 is a leap year but 2100 is not.
2. If the year is not divisible by 100, then it is a leap year if and if only it is divisible by 4. For example, 2008 is a leap year but 2009 is not.

Demonstrate the class in a program that asks the user to enter the month (letting the user enter an integer in the range of 1 through 12) and the year. The program should then display the number of days in that month. Here is a sample run of the program:

```
Enter a month (1-12): 2 [Enter]
Enter a year: 2008 [Enter]
29 days
```

### 10. A Game of Twenty-One

For this assignment, you will write a program that lets the user play against the computer in a variation of the popular blackjack card game. In this variation of the game, two six-sided dice are used instead of cards. The dice are rolled, and the player tries to beat the computer's hidden total without going over 21.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player agrees to roll the dice, and the player's total does not exceed 21.
- At the beginning of each round, the program will ask the user whether or not he or she wants to roll the dice to accumulate points.
- During each round, the program simulates the rolling of two six-sided dice. It rolls the dice first for the computer, and then it asks the user whether he or she wants to roll. (Use the `Die` class that was shown in Code Listing 6-14 to simulate the dice.)
- The loop keeps a running total of both the computer's and the user's points.
- The computer's total should remain hidden until the loop has finished.
- After the loop has finished, the computer's total is revealed, and the player with the most points, without going over 21, wins.

### 11. Freezing and Boiling Points

The following table lists the freezing and boiling points of several substances.

| Substance | Freezing Point | Boiling Point |
|---|---|---|
| Ethyl Alcohol | −173 | 172 |
| Oxygen | −362 | −306 |
| Water | 32 | 212 |

Design a class that stores a temperature in a `temperature` field and has the appropriate accessor and mutator methods for the field. In addition to appropriate constructors, the class should have the following methods:

- **`isEthylFreezing`.** This method should return the `boolean` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of ethyl alcohol. Otherwise, the method should return `false`.
- **`isEthylBoiling`.** This method should return the `boolean` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of ethyl alcohol. Otherwise, the method should return `false`.
- **`isOxygenFreezing`.** This method should return the `boolean` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of oxygen. Otherwise, the method should return `false`.
- **`isOxygenBoiling`.** This method should return the `boolean` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of oxygen. Otherwise, the method should return `false`.
- **`isWaterFreezing`.** This method should return the `boolean` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of water. Otherwise, the method should return `false`.
- **`isWaterBoiling`.** This method should return the `boolean` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of water. Otherwise, the method should return `false`.

Write a program that demonstrates the class. The program should ask the user to enter a temperature, and then display a list of the substances that will freeze at that temperature and those that will boil at that temperature. For example, if the temperature is –20 the class should report that water will freeze and oxygen will boil at that temperature.

### 12. `SavingsAccount` Class

Design a `SavingsAccount` class that stores a savings account's annual interest rate and balance. The class constructor should accept the amount of the savings account's starting balance. The class should also have methods for subtracting the amount of a withdrawal, adding the amount of a deposit, and adding the amount of monthly interest to the balance. The monthly interest rate is the annual interest rate divided by twelve. To add the monthly interest to the balance, multiply the monthly interest rate by the balance, and add the result to the balance.

Test the class in a program that calculates the balance of a savings account at the end of a period of time. It should ask the user for the annual interest rate, the starting balance, and the number of months that have passed since the account was established. A loop should then iterate once for every month, performing the following:

a. Ask the user for the amount deposited into the account during the month. Use the class method to add this amount to the account balance.
b. Ask the user for the amount withdrawn from the account during the month. Use the class method to subtract this amount from the account balance.
c. Use the class method to calculate the monthly interest.

After the last iteration, the program should display the ending balance, the total amount of deposits, the total amount of withdrawals, and the total interest earned.

### 13. Deposit and Withdrawal Files

Use Notepad or another text editor to create a text file named *Deposits.txt*. The file should contain the following numbers, one per line:

```
100.00
124.00
78.92
37.55
```

Next, create a text file named *Withdrawals.txt*. The file should contain the following numbers, one per line:

```
29.88
110.00
27.52
50.00
12.90
```

The numbers in the *Deposits.txt* file are the amounts of deposits that were made to a savings account during the month, and the numbers in the *Withdrawals.txt* file are the amounts of withdrawals that were made during the month. Write a program that creates an instance of the `SavingsAccount` class that you wrote in Programming Challenge 12. The starting balance for the object is 500.00. The program should read the values from the *Deposits.txt* file and use the object's method to add them to the account balance. The program should read the values from the *Withdrawals.txt* file and use the object's method to subtract them from the account balance. The program should call the class method to calculate the monthly interest, and then display the ending balance and the total interest earned.

### 14. Patient Charges

Write a class named `Patient` that has fields for the following data:

- First name, middle name, and last name
- Address, city, state, and ZIP code
- Phone number
- Name and phone number of emergency contact

The `Patient` class should have a constructor, accessor, and mutator that accept an argument for each field.

Next, write a class named `Procedure` that represents a medical procedure that has been performed on a patient. The `Procedure` class should have fields for the following data:

- Name of the procedure
- Date of the procedure
- Name of the practitioner who performed the procedure
- Charges for the procedure

The `Procedure` class should have a constructor, accessor, and mutator that accept an argument for each field.

Next, write a program that creates an instance of the `Patient` class, initialized with sample data. Then, create three instances of the `Procedure` class, initialized with the following data:

| **Procedure #1:** | **Procedure #2:** | **Procedure #3:** |
|---|---|---|
| Procedure name: Physical Exam | Procedure name: X-ray | Procedure name: Blood test |
| Date: Today's date | Date: Today's date | Date: Today's date |
| Practitioner: Dr. Irvine | Practitioner: Dr. Jamison | Practitioner: Dr. Smith |
| Charge: 250.00 | Charge: 500.00 | Charge: 200.00 |

The program should display the patient's information, information about all three of the procedures, and the total charges of the three procedures.

### 15. Dice Game

Write a program that uses the `Die` class that was presented in this chapter to play a simple dice game between the computer and the user. The program should create two instances of the `Die` class (each a 6-sided die). One `Die` object is the computer's die, and the other `Die` object is the user's die.

The program should have a loop that iterates 10 times. Each time the loop iterates, it should roll both dice. The die with the highest value wins. (In case of a tie, there is no winner for that particular roll of the dice.)

As the loop iterates, the program should keep count of the number of times the computer wins, and the number of times that the user wins. After the loop performs all of its iterations, the program should display who was the grand winner, the computer or the user.

### 16. Roulette Wheel Colors

On a roulette wheel, the pockets are numbered from 0 to 36. The colors of the pockets are as follows:

- Pocket 0 is green.
- For pockets 1 through 10, the odd numbered pockets are red and the even numbered pockets are black.
- For pockets 11 through 18, the odd numbered pockets are black and the even numbered pockets are red.
- For pockets 19 through 28, the odd numbered pockets are red and the even numbered pockets are black.
- For pockets 29 through 36, the odd numbered pockets are black and the even numbered pockets are red.

Write a class named `RoulettePocket`. The class's constructor should accept a pocket number. The class should have a method named `getPocketColor` that returns the pocket's color, as a string.

Demonstrate the class in a program that asks the user to enter a pocket number, and displays whether the pocket is green, red, or black. The program should display an error message if the user enters a number that is outside the range of 0 through 36.

### 17. Coin Toss Simulator

Write a class named `Coin`. The `Coin` class should have the following field:

- A `String` named `sideUp`. The `sideUp` field will hold either "heads" or "tails" indicating the side of the coin that is facing up.

The `Coin` class should have the following methods:

- A no-arg constructor that randomly determines the side of the coin that is facing up ("heads" or "tails") and initializes the `sideUp` field accordingly.
- A `void` method named `toss` that simulates the tossing of the coin. When the toss method is called, it randomly determines the side of the coin that is facing up ("heads" or "tails") and sets the `sideUp` field accordingly.
- A method named `getSideUp` that returns the value of the `sideUp` field.

Write a program that demonstrates the `Coin` class. The program should create an instance of the class and display the side that is initially facing up. Then, use a loop to toss the coin 20 times. Each time the coin is tossed, display the side that is facing up. The program should keep count of the number of times heads is facing up and the number of times tails is facing up, and display those values after the loop finishes.

### 18. Tossing Coins for a Dollar

For this assignment you will create a game program using the `Coin` class from Programming Challenge 16. The program should have three instances of the `Coin` class: one representing a quarter, one representing a dime, and one representing a nickel.

When the game begins, your starting balance is $0. During each round of the game, the program will toss the simulated coins. When a coin is tossed, the value of the coin is added to your balance if it lands heads-up. For example, if the quarter lands heads-up, 25 cents is added to your balance. Nothing is added to your balance for coins that land tails-up. The game is over when your balance reaches one dollar or more. If your balance is exactly one dollar, you win the game. You lose if your balance exceeds one dollar.

### 19. Fishing Game Simulation

For this assignment, you will write a program that simulates a fishing game. In this game, a six-sided die is rolled to determine what the user has caught. Each possible item is worth a certain number of fishing points. The points will remain hidden until the user is finished fishing, and then a message is displayed congratulating the user, depending on the number of fishing points gained.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player wants to fish for more items.
- At the beginning of each round, the program will ask the user whether or not he or she wants to continue fishing.

- The program simulates the rolling of a six-sided die (use the `Die` class that was shown in Code Listing 6-14).
- Each item that can be caught is represented by a number generated from the die; for example, 1 for “a huge fish”, 2 for “an old shoe”, 3 for “a little fish”, and so on.
- Each item the user catches is worth a different amount of points.
- The loop keeps a running total of the user’s fishing points.
- After the loop has finished, the total number of fishing points is displayed, along with a message that varies depending on the number of points earned.

CHAPTER

# 7

# Arrays and the ArrayList Class

## TOPICS

7.1 Introduction to Arrays
7.2 Processing Array Elements
7.3 Passing Arrays as Arguments to Methods
7.4 Some Useful Array Algorithms and Operations
7.5 Returning Arrays from Methods
7.6 `String` Arrays
7.7 Arrays of Objects
7.8 The Sequential Search Algorithm
7.9 Two-Dimensional Arrays
7.10 Arrays with Three or More Dimensions
7.11 The Selection Sort and the Binary Search Algorithms
7.12 Command-Line Arguments and Variable-Length Argument Lists
7.13 The `ArrayList` Class
7.14 Common Errors to Avoid

## 7.1 Introduction to Arrays

**CONCEPT: An array can hold multiple values of the same data type simultaneously.**

The primitive variables you have worked with so far are designed to hold one value at a time. Each of the variable declarations in Figure 7-1 causes only enough memory to be reserved to hold one value of the specified data type.

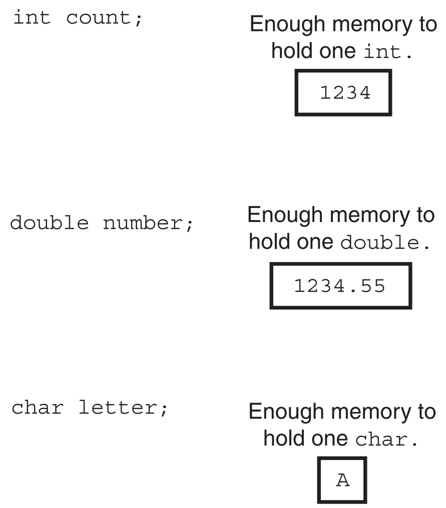
An array, however, is an object that can store a group of values, all of the same type. Creating and using an array in Java is similar to creating and using any other type of object: You declare a reference variable and use the `new` key word to create an instance of the array in memory. Here is an example of a statement that declares an array reference variable:

```
int[] numbers;
```

This statement declares `numbers` as an array reference variable. The `numbers` variable can reference an array of `int` values. Notice that this statement looks like a regular `int` variable declaration except for the set of brackets that appears after the key word `int`. The brackets indicate that this variable is a reference to an `int` array. Declaring an array reference variable does not create an array. The next step in the process is to use the `new` key word to create an array and assign its address to the `numbers` variable. The following statement shows an example:

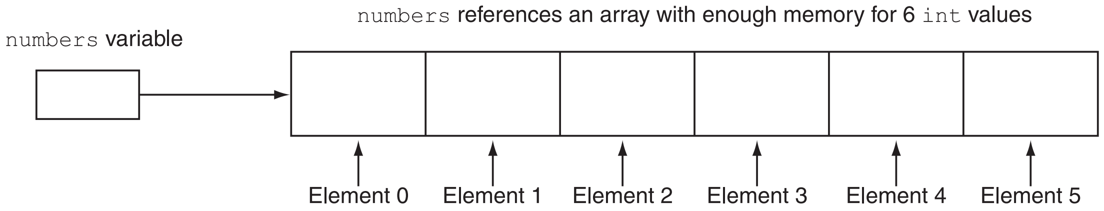
```
numbers = new int[6];
```

**Figure 7-1** Variable declarations and their memory allocations


int count;
Enough memory to hold one int.
1234
double number;
Enough memory to hold one double.
1234.55
char letter;
Enough memory to hold one char.
A

The number inside the brackets is the array's *size declarator*. It indicates the number of elements, or values, the array can hold. When this statement is executed, `numbers` will reference an array that can hold six elements, each one an `int`. This is shown in Figure 7-2.

**Figure 7-2** The `numbers` array


numbers variable
numbers references an array with enough memory for 6 int values
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

As with any other type of object, it is possible to declare a reference variable and create an instance of an array with one statement. Here is an example:

```
int[] numbers = new int[6];
```

Arrays of any data type can be declared. The following are all valid array declarations:

```
float[] temperatures = new float[100];
char[] letters = new char[41];
long[] units = new long[50];
double[] sizes = new double[1200];
```

An array's size declarator must be a non-negative integer expression. It can be a literal value, as shown in the previous examples, or a variable. It is a common practice to use a `final` variable as a size declarator. Here is an example:

```
final int NUM_ELEMENTS = 6;
int[] numbers = new int[NUM_ELEMENTS];
```

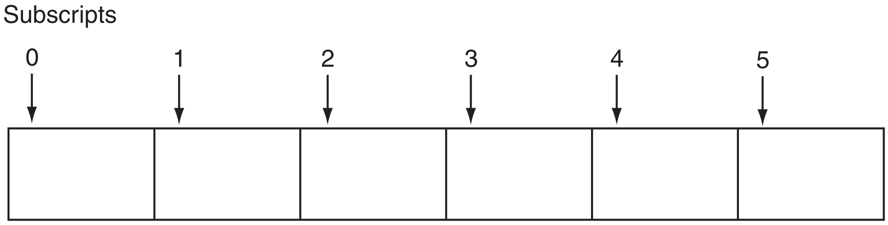
This practice can make programs easier to maintain. When we store the size of an array in a variable, we can use the variable instead of a literal number when we refer to the size of the array. If we ever need to change the array's size, we need only to change the value of the variable. The variable should be `final` so its contents cannot be changed during the program's execution.

**NOTE:** Once an array is created, its size cannot be changed.

## Accessing Array Elements

Although an array has only one name, the elements in the array may be accessed and used as individual variables. This is possible because each element is assigned a number known as a *subscript*. A subscript is used as an index to pinpoint a specific element within an array. The first element is assigned the subscript 0, the second element is assigned 1, and so forth. The six elements in the `numbers` array (described earlier) would have the subscripts 0 through 5. This is shown in Figure 7-3.

**Figure 7-3** Subscripts for the `numbers` array



The `numbers` array has six elements, numbered 0 through 5.

Subscript numbering always starts at zero. The subscript of the last element in an array is one less than the total number of elements in the array. This means that for the `numbers` array, which has six elements, 5 is the subscript for the last element.

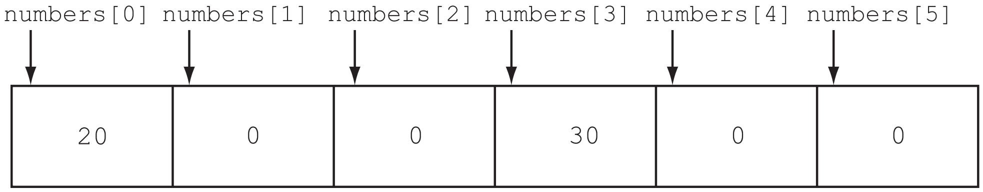
Each element in the `numbers` array, when accessed by its subscript, can be used as an `int` variable. For example, look at the following code. The first statement stores 20 in the first element of the array (element 0), and the second statement stores 30 in the fourth element (element 3).

```
numbers[0] = 20;
numbers[3] = 30;
```

**NOTE:** The expression `numbers[0]` is pronounced "numbers sub zero." You read these assignment statements as "numbers sub zero is assigned twenty" and "numbers sub three is assigned thirty."

Figure 7-4 illustrates the contents of the array after these statements execute.

**Figure 7-4** Contents of the array after 20 is assigned to `numbers[0]` and 30 is assigned to `numbers[3]`



**NOTE:** By default, Java initializes array elements with 0. In Figure 7-4, values have not been stored in elements 1, 2, 4, and 5, so they are shown as 0s.

By this point you should understand the difference between the array size declarator and a subscript. When you use the `new` key word to create an array object, the number inside the brackets is the size declarator. It indicates the number of elements in the array. The number inside the brackets in an assignment statement or any statement that works with the contents of an array is a subscript. It is used to access a specific element in the array.

## Inputting and Outputting Array Contents

You can read values from the keyboard and store them in an array element just as you can a regular variable. You can also output the contents of an array element with `print` and `println`. Code Listing 7-1 shows an array being used to store and display values entered by the user. Figure 7-5 shows the contents of the `hours` array with the values entered by the user in the example output.

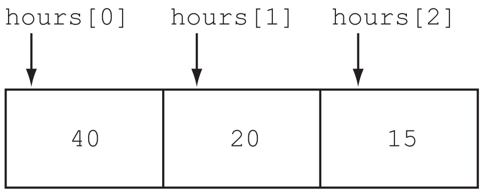
**Code Listing 7-1** (`ArrayDemo1.java`)

```java
import java.util.Scanner;

/**
   This program shows values being stored in an array's
   elements and displayed.
*/

public class ArrayDemo1
{
   public static void main(String[] args)
   {
      final int EMPLOYEES = 3;          // Number of employees
      int[] hours = new int[EMPLOYEES]; // Array of hours

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      System.out.println("Enter the hours worked by " +
                         EMPLOYEES + " employees.");

      // Get the hours worked by employee 1.
      System.out.print("Employee 1: ");
      hours[0] = keyboard.nextInt();

      // Get the hours worked by employee 2.
      System.out.print("Employee 2: ");
      hours[1] = keyboard.nextInt();

      // Get the hours worked by employee 3.
      System.out.print("Employee 3: ");
      hours[2] = keyboard.nextInt();

      // Display the values entered.
      System.out.println("The hours you entered are:");
      System.out.println(hours[0]);
      System.out.println(hours[1]);
      System.out.println(hours[2]);
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the hours worked by 3 employees.
Employee 1: 40 [Enter]
Employee 2: 20 [Enter]
Employee 3: 15 [Enter]
The hours you entered are:
40
20
15
```

**Figure 7-5** Contents of the hours array



VideoNote
Accessing Array Elements in a Loop

Subscript numbers can be stored in variables. This makes it possible to use a loop to "cycle through" an entire array, performing the same operation on each element. For example, Code Listing 7-1 could be simplified by using two `for` loops: one for inputting the values into the array and the other for displaying the contents of the array. This is shown in Code Listing 7-2.

**Code Listing 7-2** (ArrayDemo2.java)

```
import java.util.Scanner;

/**
   This program shows an array being processed with loops.
*/

public class ArrayDemo2
{
   public static void main(String[] args)
   {
      final int EMPLOYEES = 3;                   // Number of employees
      int[] hours = new int[EMPLOYEES];          // Array of hours

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      System.out.println("Enter the hours worked by " +
                         EMPLOYEES + " employees.");

      // Get the hours for each employee.
      for (int index = 0; index < EMPLOYEES; index++)
      {
         System.out.print("Employee " + (index + 1) + ": ");
         hours[index] = keyboard.nextInt();
      }

      System.out.println("The hours you entered are:");

      // Display the values entered.
      for (int index = 0; index < EMPLOYEES; index++)
        System.out.println(hours[index]);
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the hours worked by 3 employees.
Employee 1: 40 [Enter]
Employee 2: 20 [Enter]
Employee 3: 15 [Enter]
The hours you entered are:
40
20
15
```

Let's take a closer look at the first loop in this program, which appears in lines 21 through 25. Notice that the loop's control variable, `index`, is used as a subscript in line 24:

```
hours[index] = keyboard.nextInt();
```

The variable `index` starts at 0. During the loop's first iteration, the user's input is stored in `hours[0]`. Then, `index` is incremented, so its value becomes 1. During the next iteration, the user's input is stored in `hours[1]`. This continues until values have been stored in all of the elements of the array. Notice that the loop correctly starts and ends the control variable with valid subscript values (0 through 2), as illustrated in Figure 7-6. This ensures that only valid subscripts are used.

**Figure 7-6** Annotated loop

The variable `index` starts at 0, which is the first valid subscript value.

The loop ends before the variable `index` reaches 3, which is the first invalid subscript value.

```
for (int index = 0; index < EMPLOYEES; index++)
{
   System.out.print("Employee " + (index + 1) + ": ");
   hours[index] = keyboard.nextInt();
}
```

## Java Performs Bounds Checking

Java performs array bounds checking, which means that it does not allow a statement to use a subscript that is outside the range of valid subscripts for an array. For example, the following statement creates an array with 10 elements. The valid subscripts for the array are 0 through 9.

```
int[] values = new int[10];
```

Java will not allow a statement to use a subscript that is less than 0 or greater than 9 with this array. Bounds checking occurs at runtime. The Java compiler does not display an error message when it processes a statement that uses an invalid subscript. Instead, when the statement executes, the program throws an exception and immediately terminates. For instance, the program in Code Listing 7-3 declares a three-element array, but attempts to store four values in the array. In line 17, when the program attempts to store a value in `values[3]`, it halts and an error message is displayed.

**Code Listing 7-3** (`InvalidSubscript.java`)

```
1  /**
2     This program uses an invalid subscript with an array.
3  */
4
```

```
public class InvalidSubscript
{
   public static void main(String[] args)
   {
      int[] values = new int[3];

      System.out.println("I will attempt to store four " +
                         "numbers in a three-element array.");

      for (int index = 0; index < 4; index++)
      {
         System.out.println("Now processing element " + index);
         values[index] = 10;
      }
   }
}
```

**Program Output**

```
I will attempt to store four numbers in a three-element array.
Now processing element 0
Now processing element 1
Now processing element 2
Now processing element 3
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3
         at InvalidSubscript.main(InvalidSubscript.java:17)
```

**NOTE:** The error message you see may be different, depending on your system.

## Watch Out for Off-by-One Errors

Because array subscripts start at 0 rather than 1, you have to be careful not to perform an off-by-one error. For example, look at the following code:

```
// This code has an off-by-one error.
final int SIZE = 100;
int[] numbers = new int[SIZE];
for (int index = 1; index <= SIZE; index++)
   numbers[index] = 0;
```

The intent of this code is to create an array of integers with 100 elements, and store the value 0 in each element. However, this code has an off-by-one error. The loop uses its control variable, `index`, as a subscript with the `numbers` array. During the loop's execution, the variable `index` takes on the values 1 through 100, when it should take on the values 0 through 99. As a result, the first element, which is at subscript 0, is skipped. In addition, the loop attempts to use 100 as a subscript during the last iteration. Because 100 is an invalid subscript, the program will throw an exception and halt.

## Array Initialization

Like regular variables, Java allows you to initialize an array's elements when you create the array. Here is an example:

```
int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

This statement declares the reference variable `days`, creates an array in memory, and stores initial values in the array. The series of values inside the braces and separated with commas is called an initialization list. These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days[0]`, the second value, 28, is stored in `days[1]`, and so forth.) Note that you do not use the `new` key word when you use an initialization list. Java automatically creates the array and stores the values in the initialization list in it.

The Java compiler determines the size of the array by the number of items in the initialization list. Because there are 12 items in the example statement's initialization list, the array will have 12 elements. The program in Code Listing 7-4 demonstrates an array being initialized.

**Code Listing 7-4** (ArrayInitialization.java)

```
/**
   This program shows an array being initialized.
*/

public class ArrayInitialization
{
   public static void main(String[] args)
   {
      int[] days = { 31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31 };

      for (int index = 0; index < 12; index++)
      {
         System.out.println("Month " + (index + 1) +
                            " has " + days[index] +
                            " days.");
      }
   }
}
```

**Program Output**

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

Java allows you to spread the initialization list across multiple lines. Both of the following array declarations are equivalent:

```
double[] coins = { 0.05, 0.1, 0.25 };
double[] coins = { 0.05,
                   0.1,
                   0.25 };
```

## Alternate Array Declaration Notation

Java allows you to use two different styles when declaring array reference variables. The first style is the one used in this book, with the brackets immediately following the data type, as shown here:

```
int[] numbers;
```

In the second style the brackets are placed after the variable name, as shown here:

```
int numbers[];
```

Both of these statements accomplish the same thing: They declare that `numbers` is a reference to an `int` array. The difference between the two styles is noticed when more than one variable is declared in the same statement. For example, look at the following statement:

```
int[] numbers, codes, scores;
```

This statement declares three variables: `numbers`, `codes`, and `scores`. All three are references to `int` arrays. This makes perfect sense because `int[]` is the data type for all the variables declared in the statement. Now look at the following statement, which uses the alternate notation:

```
int numbers[], codes, scores;
```

This statement declares the same three variables, but only `numbers` is a reference to an `int` array. The `codes` and `scores` variables are regular `int` variables. This is because `int` is the data type for all the variables declared in the statement, and only `numbers` is followed by the brackets. To declare all three of these variables as references to `int` arrays using the alternate notation, you need to write a set of brackets after each variable name. Here is an example:

```
int numbers[], codes[], scores[];
```

The first style is the standard notation for most Java programmers, so that is the style used in this book.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

7.1 Write statements that create the following arrays:
- a) A 100-element `int` array referenced by the variable `employeeNumbers`.
- b) A 25-element `double` array referenced by the variable `payRates`.
- c) A 14-element `float` array referenced by the variable `miles`.
- d) A 1000-element `char` array referenced by the variable `letters`.

7.2 What's wrong with the following array declarations?

```
int[] readings = new int[-1];
double[] measurements = new double[4.5];
```

7.3 What would the valid subscript values be in a four-element array of `double`s?

7.4 What is the difference between an array's size declarator and a subscript?

7.5 What does it mean for a subscript to be out-of-bounds?

7.6 What happens in Java when a program tries to use a subscript that is out-of-bounds?

7.7 What is the output of the following code?

```
int[] values = new int[5];
for (int count = 0; count < 5; count++)
     values[count] = count + 1;
for (int count = 0; count < 5; count++)
     System.out.println(values[count]);
```

7.8 Write a statement that creates and initializes a `double` array with the following values: 1.7, 6.4, 8.9, 3.1, and 9.2. How many elements are in the array?

## 7.2 Processing Array Elements

**CONCEPT: Individual array elements are processed like any other type of variable.**

Processing array elements is no different from processing other variables. For example, the following statement multiplies `hours[3]` by the variable `payRate`:

```
grossPay = hours[3] * payRate;
```

The following are examples of pre-increment and post-increment operations on array elements:

```
int[] score = {7, 8, 9, 10, 11};
++score[2];     // Pre-increment operation
score[4]++;     // Post-increment operation
```

When using increment and decrement operators, be careful not to use the operator on the subscript when you intend to use it on the array element. For example, the following statement decrements the variable `count`, but does nothing to the value stored in the array element `amount[count]`:

```
amount[count--];
```

Code Listing 7-5 demonstrates the use of array elements in a simple mathematical statement. A loop steps through each element of the array, using the elements to calculate the gross pay of five employees.

**Code Listing 7-5** `(PayArray.java)`

```java
import java.util.Scanner;

/**
   This program stores in an array the hours worked by
   five employees who all make the same hourly wage.
*/

public class PayArray
{
   public static void main(String[] args)
   {
      final int EMPLOYEES = 5;  // Number of employees
      double payRate;           // Hourly pay rate
      double grossPay;          // Gross pay

      // Create an array to hold employee hours.
      int[] hours = new int[EMPLOYEES];

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the hours worked by each employee.
      System.out.println("Enter the hours worked by " +
                         EMPLOYEES + " employees who all earn " +
                         "the same hourly rate.");

     for (int index = 0; index < EMPLOYEES; index++)
     {
        System.out.print( "Employee #" + (index + 1) + ": ");
        hours[index] = keyboard.nextInt();
     }

     // Get the hourly pay rate.
     System.out.print("Enter the hourly rate for each employee: ");
     payRate = keyboard.nextDouble();

     // Display each employee's gross pay.
     System.out.println( "Here is each employee's gross pay:");
     for (int index = 0; index < EMPLOYEES; index++)
     {
        grossPay = hours[index] * payRate;
42          System.out.println("Employee #" + (index + 1) +
43                             ": $" + grossPay);
44        }
45     }
46  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the hours worked by 5 employees who all earn the same hourly rate.
Employee #1: 10 [Enter]
Employee #2: 20 [Enter]
Employee #3: 30 [Enter]
Employee #4: 40 [Enter]
Employee #5: 50 [Enter]
Enter the hourly rate for each employee: 10 [Enter]
Here is each employee's gross pay:
Employee #1: $100.0
Employee #2: $200.0
Employee #3: $300.0
Employee #4: $400.0
Employee #5: $500.0
```

In line 41, the following statement assigns the value of `hours[index]` times `payRate` to the `grossPay` variable:

```
grossPay = hours[index] * payRate;
```

Array elements may also be used in relational expressions. For example, the following `if` statement determines whether `cost[20]` is less than `cost[0]`:

```
if (cost[20] < cost[0])
```

And the following `while` loop iterates as long as `value[count]` does not equal 0:

```
while (value[count] != 0)
{
   Statements
}
```

In this chapter's source code (available at www.pearsonhighered.com/gaddis), you will find the file *Overtime.java*, which is a modification of the *PayArray.java* program in Code Listing 7-5. The *Overtime.java* program includes overtime wages in the gross pay. If an employee works more than 40 hours, an overtime pay rate of 1.5 times the regular pay rate is used for the excess hours.

## Array Length

Each array in Java has a public field named `length`. This field contains the number of elements in the array. For example, consider an array created by the following statement:

```
double[] temperatures = new double[25];
```

Because the `temperatures` array has 25 elements, the following statement would assign 25 to the variable `size`:

```
size = temperatures.length;
```

The `length` field can be useful when processing the entire contents of an array. For example, the following loop steps through an array and displays the contents of each element. The array's `length` field is used in the test expression as the upper limit for the loop control variable:

```
for (int i = 0; i < temperatures.length; i++)
   System.out.println(temperatures[i]);
```

**WARNING!** Be careful not to cause an off-by-one error when using the `length` field as the upper limit of a subscript. The `length` field contains the number of elements in an array. The largest subscript in an array is `length - 1`.

**NOTE:** You cannot change the value of an array's `length` field.

## The Enhanced `for` Loop

Java provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing. It is known as the *enhanced* ***`for`*** *loop*. Here is the general format of the enhanced `for` loop:

```
for (dataType elementVariable : array)
  statement;
```

The enhanced `for` loop is designed to iterate once for every element in an array. Each time the loop iterates, it copies an array element to a variable. Let's look at the syntax more closely as follows:

- *`dataType elementVariable`* is a variable declaration. This variable will receive the value of a different array element during each loop iteration. During the first loop iteration, it receives the value of the first element; during the second iteration, it receives the value of the second element, and so on. This variable must be of the same data type as the array elements, or a type that the elements can automatically be converted to.
- *`array`* is the name of an array on which you wish the loop to operate. The loop will iterate once for every element in the array.
- *`statement`* is a statement that executes during a loop iteration.

For example, assume that we have the following array declaration:

```
int[] numbers = { 3, 6, 9 };
```

We can use the following enhanced `for` loop to display the contents of the `numbers` array:

```
for (int val : numbers)
   System.out.println(val);
```

Because the `numbers` array has three elements, this loop will iterate three times. The first time it iterates, the `val` variable will receive the value in `numbers[0]`. During the second iteration, `val` will receive the value in `numbers[1]`. During the third iteration, `val` will receive the value in `numbers[2]`. The code's output will be as follows:

```
3
6
9
```

If you need to execute more than one statement in the enhanced `for` loop, simply enclose the block of statements in a set of braces. Here is an example:

```
int[] numbers = { 3, 6, 9 };
for (int val : numbers)
{
   System.out.print("The next value is ");
   System.out.println(val);
}
```

This code will produce the following output:

```
The next value is 3
The next value is 6
The next value is 9
```

### The Enhanced `for` Loop versus the Traditional `for` Loop

When you need to access the values that are stored in an array, from the first element to the last element, the enhanced `for` loop is simpler to use than the traditional `for` loop. With the enhanced `for` loop you do not have to be concerned about the size of the array, and you do not have to create an "index" variable to hold subscripts. However, there are circumstances in which the enhanced `for` loop is not adequate. You cannot use the enhanced `for` loop as follows:

- if you need to change the contents of an array element
- if you need to work through the array elements in reverse order
- if you need to access some of the array elements, but not all of them
- if you need to simultaneously work with two or more arrays within the loop
- if you need to refer to the subscript number of a particular element

In any of these circumstances, you should use the traditional `for` loop to process the array.

## Letting the User Specify an Array's Size

Java allows you to use an integer variable to specify an array's size declarator. This makes it possible to allow the user to specify an array's size. Code Listing 7-6 demonstrates this, as well as the use of the `length` field. It stores a number of test scores in an array and then displays them.

**Code Listing 7-6** `(DisplayTestScores.java)`

```
import java.util.Scanner;

/**
   This program demonstrates how the user may specify an
   array's size.
*/

public class DisplayTestScores
{
   public static void main(String[] args)
   {
      int numTests;     // The number of tests
      int[] tests;      // Array of test scores

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of test scores.
      System.out.print("How many tests do you have? ");
      numTests = keyboard.nextInt();

      // Create an array to hold that number of scores.
      tests = new int[numTests];

      // Get the individual test scores.
      for (int index = 0; index < tests.length; index++)
      {
         System.out.print("Enter test score " +
                          (index + 1) + ": ");
         tests[index] = keyboard.nextInt();
      }

      // Display the test scores.
      System.out.println();
      System.out.println("Here are the scores you entered:");
      for (int index = 0; index < tests.length; index++)
         System.out.print(tests[index] + " ");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
How many tests do you have? 5 [Enter]
Enter test score 1: 72 [Enter]
Enter test score 2: 85 [Enter]
Enter test score 3: 81 [Enter]
Enter test score 4: 94 [Enter]
Enter test score 5: 99 [Enter]

Here are the scores you entered:
72 85 81 94 99
```

This program allows the user to determine the size of the array. In line 23 the following statement creates the array, using the `numTests` variable to determine its size:

```
tests = new int[numTests];
```

The program then uses two `for` loops. The first, in lines 26 through 31, allows the user to input each test score. The second, in lines 36 and 37, displays all of the test scores. Both loops use the `length` member to control their number of iterations as follows:

```
for (int index = 0; index < tests.length; index++)
```

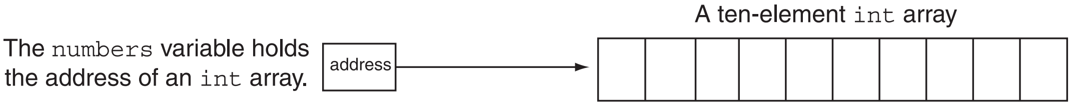
## Reassigning Array Reference Variables

It is possible to reassign an array reference variable to a different array, as demonstrated by the following code:

```
// Create an array referenced by the numbers variable.
int[] numbers = new int[10];
// Reassign numbers to a new array.
numbers = new int[5];
```

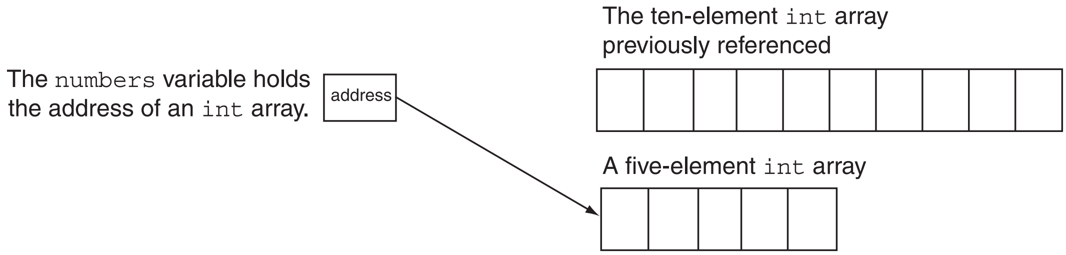
The first statement creates a ten-element integer array and assigns its address to the `numbers` variable. This is illustrated in Figure 7-7.

**Figure 7-7** The `numbers` variable references a ten-element array



The second statement allocates a five-element integer array and assigns its address to the `numbers` variable. The address of the five-element array takes the place of the address of the ten-element array. After this statement executes, the `numbers` variable references the five-element array instead of the ten-element array. This is illustrated in Figure 7-8. Because the ten-element array is no longer referenced, it cannot be accessed.

**Figure 7-8** The `numbers` variable references a five-element array


The ten-element int array
previously referenced
The numbers variable holds
the address of an int array.
address
A five-element int array

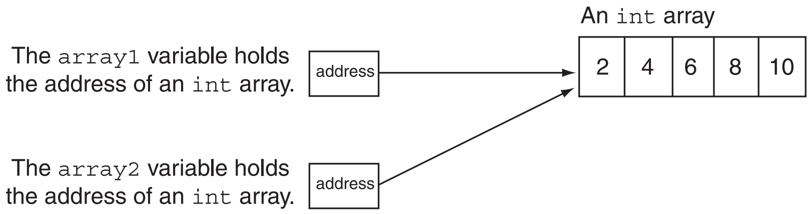
## Copying Arrays

Because an array is an object, there is a distinction between an array and the variable that references it. The array and the reference variable are two separate entities. This is important to remember when you wish to copy the contents of one array to another. You might be tempted to write something like the following code, thinking that you are copying an array:

```
int[] array1 = { 2, 4, 6, 8, 10 };
int[] array2 = array1;    // This does not copy array1.
```

The first statement creates an array and assigns its address to the `array1` variable. The second statement assigns `array1` to `array2`. This does not make a copy of the array referenced by `array1`. Rather, it makes a copy of the address that is stored in `array1` and stores it in `array2`. After this statement executes, both the `array1` and `array2` variables will reference the same array. This type of assignment operation is called a *reference copy*. Only the address of the array object is copied, not the contents of the array object. This is illustrated in Figure 7-9.

**Figure 7-9** Both `array1` and `array2` reference the same array


An int array
2
4
6
8
10
The array1 variable holds
the address of an int array.
address
The array2 variable holds
the address of an int array.
address

Code Listing 7-7 demonstrates the assigning of an array's address to two reference variables. Regardless of which variable the program uses, it is working with the same array.

**Code Listing 7-7** (`SameArray.java`)

```
/**
   This program demonstrates that two variables can
   reference the same array.
*/

public class SameArray
{
   public static void main(String[] args)
   {
      int[] array1 = { 2, 4, 6, 8, 10 };
      int[] array2 = array1;

      // Change one of the elements using array1.
      array1[0] = 200;

      // Change one of the elements using array2.
      array2[4] = 1000;

      // Display all the elements using array1
      System.out.println("The contents of array1:");
      for (int value : array1)
         System.out.print(value + " ");
      System.out.println();

      // Display all the elements using array2
      System.out.println("The contents of array2:");
      for (int value : array2)
         System.out.print(value + " ");
      System.out.println();
   }
}
```

**Program Output**

```
The contents of array1:
200 4 6 8 1000
The contents of array2:
200 4 6 8 1000
```

The program in Code Listing 7-7 illustrates that you cannot copy an array by merely assigning one array reference variable to another. Instead, to copy an array you need to copy the individual elements of one array to another. Usually, this is best done with a loop, such as the following:

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = new int[5];

for (int index = 0; index < firstArray.length; index++)
   secondArray[index] = firstArray[index];
```

The loop in this code copies each element of `firstArray` to the corresponding element of `secondArray`.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

7.9 Look at the following statements:

```
int[] numbers1 = { 1, 3, 6, 9 };
int[] numbers2 = { 2, 4, 6, 8 };
int result;
```

Write a statement that multiplies element 0 of the `numbers1` array by element 3 of the `numbers2` array and assigns the result to the `result` variable.

7.10 A program uses a variable named `array` that references an array of integers. You do not know the number of elements in the array. Write a `for` loop that stores −1 in each element of the array.

7.11 A program has the following declaration:

```
double[] values;
```

Write code that asks the user for the size of the array and then creates an array of the specified size, referenced by the `values` variable.

7.12 Look at the following statements:

```
int[] a = { 1, 2, 3, 4, 5, 6, 7 };
int[] b = new int[7];
```

Write code that copies the a array to the b array.

## 7.3 Passing Arrays as Arguments to Methods

**CONCEPT: An array can be passed as an argument to a method. To pass an array, you pass the value in the variable that references the array.**

VideoNote
Passing an Array to a Method

Quite often you'll want to write methods that process the data in arrays. As you will see, methods can be written to store values in an array, display an array's contents, total all of an array's elements, calculate their average, and so forth. Usually, such methods accept an array as an argument.

When a single element of an array is passed to a method, it is handled like any other variable. For example, Code Listing 7-8 shows a loop that passes each element of the array `numbers` to the method `showValue`.

**Code Listing 7-8** `(PassElements.java)`

```java
/**
   This program demonstrates passing individual array
   elements as arguments to a method.
*/

public class PassElements
{
   public static void main(String[] args)
   {
      int[] numbers = {5, 10, 15, 20, 25, 30, 35, 40};

      for (int index = 0; index < numbers.length; index++)
         showValue(numbers[index]);
   }

   /**
      The showValue method displays its argument.
      @param n The value to display.
   */

   public static void showValue(int n)
   {
      System.out.print(n + " ");
   }
}
```

**Program Output**

```
5 10 15 20 25 30 35 40
```

Each time `showValue` is called in this program, an array element is passed to the method. The `showValue` method has an `int` parameter variable named `n`, which receives the argument. The method simply displays the contents of `n`. If the method were written to accept the entire array as an argument, however, the parameter would have to be set up differently. For example, consider the following method definition. The parameter `array` is declared as an array reference variable. This indicates that the argument will be an array, not a single value.

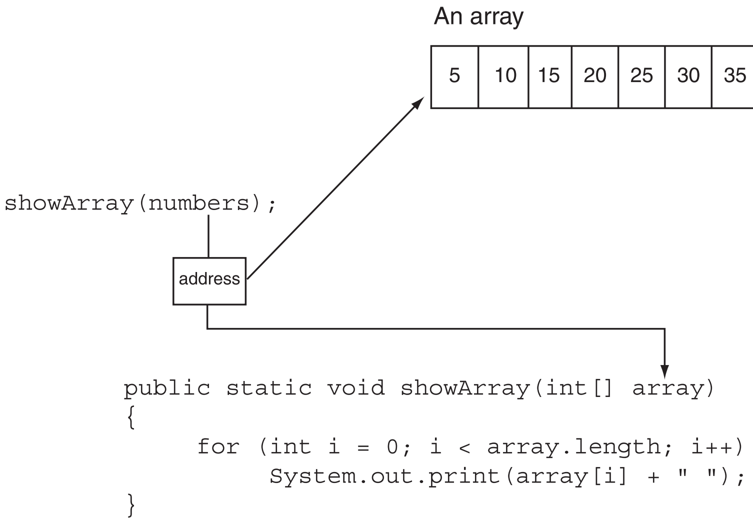
```java
public static void showArray(int[] array)
{
   for (int i = 0; i < array.length; i++)
      System.out.print(array[i] + " ");
}
```

When you pass an array as an argument, you simply pass the value in the variable that references the array, as shown here:

```
showArray(numbers);
```

When an entire array is passed into a method, it is passed just as an object is passed: The actual array itself is not passed, but a reference to the array is passed into the parameter. Consequently, this means the method has direct access to the original array. This is illustrated in Figure 7-10.

**Figure 7-10** An array passed as an argument



```
showArray(numbers);
```

```
public static void showArray(int[] array)
{
     for (int i = 0; i < array.length; i++)
          System.out.print(array[i] + " ");
}
```

Code Listing 7-9 shows the `showArray` method in use, as well as another method, `getValues`. The `getValues` method accepts an array as an argument. It asks the user to enter a value for each element.

**Code Listing 7-9** **(PassArray.java)**

```
import java.util.Scanner;

/**
   This program demonstrates passing an array
   as an argument to a method.
*/

public class PassArray
{
   public static void main(String[] args)
   {
      final int ARRAY_SIZE = 4;  // Size of the array

      // Create an array.
      int[] numbers = new int[ARRAY_SIZE];

      // Pass the array to the getValues method.
      getValues(numbers);

      System.out.println("Here are the " +
                    "numbers that you entered:");

      // Pass the array to the showArray method.
      showArray(numbers);
   }

   /**
      The getValues method accepts a reference
      to an array as its argument. The user is
      asked to enter a value for each element.
      @param array A reference to the array.
   */

   private static void getValues(int[] array)
   {
      // Create a Scanner objects for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      System.out.println("Enter a series of " +
                    array.length + " numbers.");

      // Read values into the array
      for (int index = 0; index < array.length; index++)
      {
         System.out.print("Enter number " +
                          (index + 1) + ": ");
         array[index] = keyboard.nextInt();
      }
   }

   /**
      The showArray method accepts an array as
      an argument and displays its contents.
      @param array A reference to the array.
   */

   public static void showArray(int[] array)
   {
      // Display the array elements.
      for (int index = 0; index < array.length; index++)
         System.out.print(array[index] + " ");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter a series of 4 numbers.
Enter number 1: 2 [Enter]
Enter number 2: 4 [Enter]
Enter number 3: 6 [Enter]
Enter number 4: 8 [Enter]
Here are the numbers that you entered:
2 4 6 8
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

7.13 Look at the following method header:

```
public static void myMethod(double[] array)
```

Here is an array declaration:

```
double[] numbers = new double[100];
```

Write a statement that passes the `numbers` array to the `myMethod` method.

7.14 Write a method named `zero`, which accepts an `int` array as an argument and stores the value 0 in each element.

## 7.4 Some Useful Array Algorithms and Operations

### Comparing Arrays

In the previous section you saw that you cannot copy an array by simply assigning its reference variable to another array's reference variable. In addition, you cannot use the == operator to compare two array reference variables and determine whether the arrays are equal. For example, the following code appears to compare two arrays, but in reality does not:

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = { 5, 10, 15, 20, 25 };
if (firstArray == secondArray)       // This is a mistake.
   System.out.println("The arrays are the same.");
else
   System.out.println("The arrays are not the same.");
```

When you use the == operator with reference variables, including those that reference arrays, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables. Because the two array variables in this code reference different objects in memory, they will contain different addresses. Therefore, the result of the `boolean` expression `firstArray == secondArray` is `false` and the code reports that the arrays are not the same.

To compare the contents of two arrays, you must compare the elements of the two arrays.

For example, look at the following code:

```
int[] firstArray = { 2, 4, 6, 8, 10 };
int[] secondArray = { 2, 4, 6, 8, 10 };
boolean arraysEqual = true;    // Flag variable
int index = 0;                 // Loop control variable

// First determine whether the arrays are the same size.
if (firstArray.length != secondArray.length)
   arraysEqual = false;

// Next determine whether the elements contain the same data.
while (arraysEqual && index < firstArray.length)
{
   if (firstArray[index] != secondArray[index])
      arraysEqual = false;
   index++;
}

if (arraysEqual)
   System.out.println("The arrays are equal.");
else
  System.out.println("The arrays are not equal.");
```

This code determines whether `firstArray` and `secondArray` contain the same values. A `boolean` flag variable, `arraysEqual`, which is initialized to `true`, is used to signal whether the arrays are equal. Another variable, `index`, which is initialized to 0, is used as a loop control variable.

First, this code determines whether the two arrays are the same length. If they are not the same length, then the arrays cannot be equal, so the flag variable `arraysEqual` is set to `false`. Then a `while` loop begins. The loop executes as long as `arraysEqual` is `true` and the control variable `index` is less than `firstArray.length`. During each iteration, it compares a different set of corresponding elements in the arrays. When it finds two corresponding elements that have different values, the flag variable `arraysEqual` is set to `false`. After the loop finishes, an `if` statement examines the `arraysEqual` variable. If the variable is `true`, then the arrays are equal and a message indicating so is displayed. Otherwise, they are not equal, so a different message is displayed.

## Summing the Values in a Numeric Array

To sum the values in an array you must use a loop with an accumulator variable. The loop adds the value in each array element to the accumulator. For example, assume that the following statement appears in a program and that values have been stored in the `units` array:

```
int[] units = new int[25];
```

The following loop adds the values of each element of the `units` array to the `total` variable. When the code is finished, `total` will contain the sum of all of the `units` array's elements.

```
int total = 0;       // Initialize accumulator
for (int index = 0;  index < units.length; index++)
  total += units[index];
```

## Getting the Average of the Values in a Numeric Array

The first step in calculating the average of all the values in an array is to sum the values. The second step is to divide the sum by the number of elements in the array. Assume that the following statement appears in a program and that values have been stored in the `scores` array:

```
double[] scores = new double[10];
```

The following code calculates the average of the values in the `scores` array. When the code completes, the average will be stored in the `average` variable.

```
double total = 0;    // Initialize accumulator
double average;      // Will hold the average
for (int index = 0; index < scores.length; index++)
   total += scores[index];
average = total / scores.length;
```

Notice that the last statement, which divides `total` by `scores.length`, is not inside the loop. This statement should execute only once, after the loop has finished its iterations.

## Finding the Highest and Lowest Values in a Numeric Array

The algorithms for finding the highest and lowest values in an array are very similar. First, let's look at code for finding the highest value in an array. Assume that the following statement exists in a program and that values have been stored in the `numbers` array:

```
int[] numbers = new int[50];
```

The code to find the highest value in the array is as follows:

```
int highest = numbers[0];
for (int index = 1; index < numbers.length; index++)
{
   if (numbers[index] > highest)
      highest = numbers[index];
}
```

First we copy the value in the first array element to the variable `highest`. Then the loop compares all of the remaining array elements, beginning at subscript 1, to the value in `highest`. Each time it finds a value in the array that is greater than `highest`, it copies that value to `highest`. When the loop has finished, `highest` will contain the highest value in the array.

The following code finds the lowest value in the array. As you can see, it is nearly identical to the code for finding the highest value.

```
int lowest = numbers[0];
for (int index = 1; index < numbers.length; index++)
{
   if (numbers[index] < lowest)
      lowest = numbers[index];
}
```

When the loop has finished, `lowest` will contain the lowest value in the array.

## The `SalesData` Class

To demonstrate these algorithms, look at the `SalesData` class shown in Code Listing 7-10. An instance of the class keeps sales amounts for any number of days in an array, which is a private field. Public methods are provided that return the total, average, highest, and lowest amounts of sales. The program in Code Listing 7-11 demonstrates the class, and Figure 7-11 shows an example of interaction with the program.

**Code Listing 7-10** (`SalesData.java`)

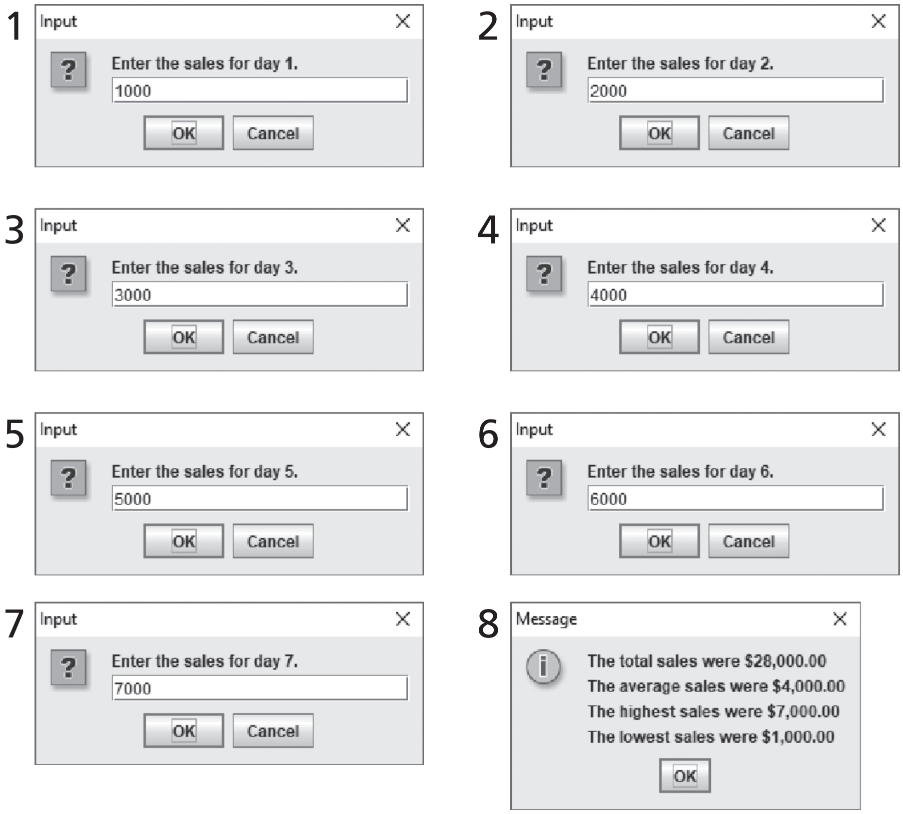
```
/**
   This class keeps the sales figures for a number of
   days in an array and provides methods for getting
   the total and average sales, and the highest and
   lowest amounts of sales.
*/

public class SalesData
{
   private double[] sales; // The sales data

   /**
      The constructor copies the elements in
      an array to the sales array.
      @param s The array to copy.
   */

   public SalesData(double[] s)
   {
      // Create an array as large as s.
      sales = new double[s.length];

      // Copy the elements from s to sales.
      for (int index = 0; index < s.length; index++)
         sales[index] = s[index];
   }

   /**
      getTotal method
      @return The total of the elements in
      the sales array.
   */

   public double getTotal()
   {
      double total = 0.0;      // Accumulator

      // Accumulate the sum of the elements
      // in the sales array.
      for (int index = 0; index < sales.length; index++)
         total += sales[index];

      // Return the total.
      return total;
   }

   /**
      getAverage method
      @return The average of the elements
      in the sales array.
   */

   public double getAverage()
   {
      return getTotal() / sales.length;
   }

   /**
      getHighest method
      @return The highest value stored
      in the sales array.
   */

   public double getHighest()
   {
      double highest = sales[0];

      for (int index = 1; index < sales.length; index++)
      {
         if (sales[index] > highest)
            highest = sales[index];
      }

      return highest;
   }

   /**
      getLowest method
      @return The lowest value stored
      in the sales array.
   */

   public double getLowest()
   {
      double lowest = sales[0];

      for (int index = 1; index < sales.length; index++)
      {
         if (sales[index] < lowest)
            lowest = sales[index];
      }

      return lowest;
   }
}
```

**Code Listing 7-11** (`Sales.java`)

```
import javax.swing.JOptionPane;

/**
   This program gathers sales amounts for the week.
   It uses the SalesData class to display the total,
   average, highest, and lowest sales amounts.
*/

public class Sales
{
   public static void main(String[] args)
   {
      final int ONE_WEEK = 7; // Number of elements

      // Create an array to hold sales amounts for a week.
      double[] sales = new double[ONE_WEEK];
```

```

     // Get the week's sales figures.
     getValues(sales);

     // Create a SalesData object, initialized
     // with the week's sales figures.
     SalesData week = new SalesData(sales);

     // Display the total, average, highest, and lowest
     // sales amounts for the week.
     JOptionPane.showMessageDialog(null,
        String.format("The total sales were $%,.2f\n" +
                      "The average sales were $%,.2f\n" +
                      "The highest sales were $%,.2f\n" +
                      "The lowest sales were $%,.2f",
                      week.getTotal(),
                      week.getAverage(),
                      week.getHighest(),
                      week.getLowest()));

     System.exit(0);
   }

   /**
      The getValues method asks the user to enter sales
      amounts for each element of an array.
      @param array The array to store the values in.
   */

   private static void getValues(double[] array)
   {
      String input; // To hold user input.

      // Get sales for each day of the week.
      for (int i = 0; i < array.length; i++)
      {
         input = JOptionPane.showInputDialog(
                  "Enter the sales for day " + (i + 1) + ".");
         array[i] = Double.parseDouble(input);
      }
   }
}
```

**Figure 7-11** Interaction with the `Sales.java` program (Oracle Corporate Counsel)



1 Enter the sales for day 1. 1000

2 Enter the sales for day 2. 2000

3 Enter the sales for day 3. 3000

4 Enter the sales for day 4. 4000

5 Enter the sales for day 5. 5000

6 Enter the sales for day 6. 6000

7 Enter the sales for day 7. 7000

8
```
The total sales were $28,000.00
The average sales were $4,000.00
The highest sales were $7,000.00
The lowest sales were $1,000.00
```

## In the Spotlight:

### Creating an Object That Processes an Array

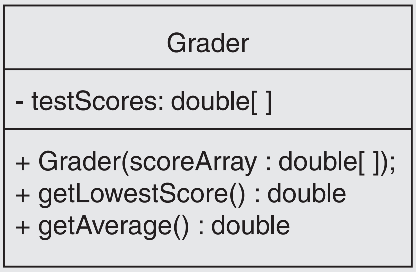
Dr. LaClaire gives a set of exams during the semester in her chemistry class. At the end of the semester, she drops each student's lowest test score before averaging the scores. She has asked you to write a program that will read a student's test scores as input and calculate the average with the lowest score dropped.

The following pseudocode shows the steps for calculating the average of a set of test scores, with the lowest score dropped:

```
Calculate the total of the scores.
Find the lowest score.
Subtract the lowest score from the total. This gives the adjusted total.
Divide the adjusted total by (number of scores – 1). This is the average.
```

You decide to create a class named `Grader`, with a constructor that accepts a double array of test scores. The `Grader` class will have a method named `getLowestScore` that returns the lowest score in the array, and a method named `getAverage` that returns the average of the test scores with the lowest score dropped. Figure 7-12 shows a UML diagram for the class.

**Figure 7-12** UML diagram for the `Grader` class


Grader
- testScores: double[ ]
+ Grader(scoreArray : double[ ]);
+ getLowestScore() : double
+ getAverage() : double

Code Listing 7-12 shows the code for the class.

### Code Listing 7-12 (Grader.java)

```
/**
   The Grader class calculates the average
   of an array of test scores, with the
   lowest score dropped.
*/

public class Grader
{
   // The testScores field is a variable
   // that will reference an array
   // of test scores.
   private double[] testScores;

   /**
      Constructor
      @param scoreArray An array of test scores.
   */

   public Grader(double[] scoreArray)
   {
      // Assign the array argument to
      // the testScores field.
      testScores = scoreArray;
   }

   /**
      getLowestScore method
      @return The lowest test score.
   */

   public double getLowestScore()
   {
      double lowest;    // To hold the lowest score

      // Get the first test score in the array.
      lowest = testScores[0];

      // Step through the rest of the array. When
      // a value less than lowest is found, assign
      // it to lowest.
      for (int index = 1; index < testScores.length; index++)
      {
          if (testScores[index] < lowest)
             lowest = testScores[index];
      }

      // Return the lowest test score.
      return lowest;
   }

   /**
      getAverage method
      @return The average of the test scores
              with the lowest score dropped.
   */

   public double getAverage()
   {
      double total = 0;   // To hold the score total
      double lowest;      // To hold the lowest score
      double average;     // To hold the average

      // If the array contains less than two test
      // scores, display an error message and set
      // average to 0.
      if (testScores.length < 2)
      {
         System.out.println("ERROR: You must have at " +
                            "least two test scores!");
         average = 0;
      }
      else
      {
         // First, calculate the total of the scores.
         for (double score : testScores)
            total += score;

         // Next, get the lowest score.
         lowest = getLowestScore();

         // Subtract the lowest score from the total.
         total -= lowest;
```

```
83
84             // Get the adjusted average.
85             average = total / (testScores.length - 1);
86          }
87
88          // Return the adjusted average.
89          return average;
90       }
91    }
```

- Line 12 declares a field named `testScores`, which will be used to reference a `double` array of test scores.
- The constructor appears in lines 19 through 24. It accepts a `double` array as an argument, which is assigned to the `testScores` field.
- The `getLowestScore` method appears in lines 31 through 49. It finds the lowest value in the `testScores` array and returns that value.
- The `getAverage` method appears in lines 57 through 90. This method first determines whether there are less than 2 elements in the `testScores` array (in line 66). If that is the case, we cannot drop the lowest score, so an error message is displayed and the `average` variable is set to 0. Otherwise, the code in lines 74 through 85 calculates the average of the test scores with the lowest score dropped, and assigns that value to the `average` variable. Line 89 returns the value of the `average` variable.

Code Listing 7-13 shows the program that Dr. LaClaire will use to calculate a student's adjusted average. The program gets a series of test scores, stores those scores in an array, and uses an instance of the `Grader` class to calculate the average.

**Code Listing 7-13** (`CalcAverage.java`)

```
import java.util.Scanner;

/**
   This program gets a set of test scores and
   uses the Grader class to calculate the average
   with the lowest score dropped.
*/

public class CalcAverage
{
   public static void main(String[] args)
   {
      int numScores;      // To hold the number of scores

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of test scores.
      System.out.print("How many test scores do you have? ");
      numScores = keyboard.nextInt();

      // Create an array to hold the test scores.
       double[] scores = new double[numScores];

      // Get the test scores and store them
      // in the scores array.
      for (int index = 0; index < numScores; index++)
      {
         System.out.print("Enter score #" +
                          (index + 1) + ": ");
         scores[index] = keyboard.nextDouble();
      }

      // Create a Grader object, passing the
      // scores array as an argument to the
      // constructor.
      Grader myGrader = new Grader(scores);

      // Display the adjusted average.
      System.out.println("Your adjusted average is " +
                         myGrader.getAverage());

      // Display the lowest score.
      System.out.println("Your lowest test score was " +
                         myGrader.getLowestScore());

   }
}
```

**Program Output with Example Input Shown in Bold**

```
How many test scores do you have?  4 [Enter]
Enter score #1: 100 [Enter]
Enter score #2: 100 [Enter]
Enter score #3: 40 [Enter]
Enter score #4: 100 [Enter]
Your adjusted average is 100.0
Your lowest test score was 40.0
```

## Partially Filled Arrays

Sometimes you need to store a series of items in an array, but you do not know the number of items that there are. As a result, you do not know the exact number of elements needed for the array. One solution is to make the array large enough to hold the largest possible number of items. This can lead to another problem, however. If the actual number of items stored in the array is less than the number of elements, the array will be only partially filled. When you process a partially filled array, you must process only the elements that contain valid data items.

A partially filled array is normally used with an accompanying integer variable that holds the number of items stored in the array. For example, suppose a program uses the following code to create an array with 100 elements, and an `int` variable named `count`, which will hold the number of items stored in the array:

```
final int ARRAY_SIZE = 100;
int[] array = new int[ARRAY_SIZE];
int count = 0;
```

Each time we add an item to the array, we must increment `count`. The following code demonstrates:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number or -1 to quit: ");
number = keyboard.nextInt();
while (number != -1 && count < array.length)
{
   array[count] = number;
   count++;
   System.out.print("Enter a number or -1 to quit: ");
   number = keyboard.nextInt();
}
```

Each iteration of this sentinel-controlled loop allows the user to enter a number to be stored in the array, or −1 to quit. The `count` variable is used as the subscript of the next available element in the array, and then incremented. When the user enters −1, or `count` reaches the size of the array, the loop stops. The following code displays all of the valid items in the array:

```
for (int index = 0; index < count; index++)
{
   System.out.println(array[index]);
}
```

Notice that this code uses `count` to determine the maximum array subscript to use.

**NOTE:** If a partially filled array is passed as an argument to a method, the variable that holds the count of items in the array must also be passed as an argument. Otherwise, the method will not be able to determine the number of items that are stored in the array.

## Working with Arrays and Files

Saving the contents of an array to a file is a straightforward procedure: Use a loop to step through each element of the array, writing its contents to the file. For example, assume a program declares an array as follows:

```
int[] numbers = { 10, 20, 30, 40, 50 };
```

The following code opens a file named *Values.txt* and writes the contents of each element of the `numbers` array to the file:

```
int[] numbers = { 10, 20, 30, 40, 50 };
```

```
// Open the file.
PrintWriter outputFile = new PrintWriter("Values.txt");

// Write the array elements to the file.
for (int index = 0; index < numbers.length; index++)
   outputFile.println(numbers[index]);
// Close the file.
outputFile.close();
```

The following code demonstrates how to open the *Values.txt* file and read its contents back into the `numbers` array.

```
final int SIZE = 5;
int[] numbers = new int[SIZE];
int index = 0;    // Loop control variable

// Open the file.
File file = new File("Values.txt");
Scanner inputFile = new Scanner(file);

// Read the file contents into the array.
while (inputFile.hasNext() && index < numbers.length)
{
   numbers[index] = inputFile.nextInt();
   index++;
}

// Close the file.
inputFile.close();
```

The file is opened, then a `while` loop reads all of the values from the file into the `numbers` array. The loop repeats as long as `inputFile.hasNext()` returns `true`, and `index` is less than `numbers.length`. The `inputFile.hasNext()` method is called to make sure there is a value remaining in the file. This prevents an error in case the file does not contain enough values to fill the array. The second condition (`index < numbers.length`) prevents the loop from writing outside the array boundaries.

## 7.5 Returning Arrays from Methods

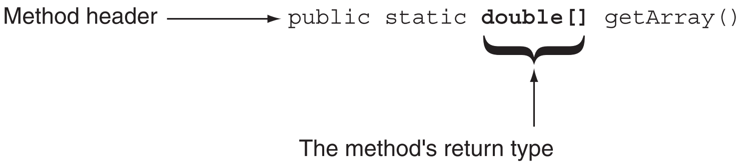
**CONCEPT: In addition to accepting arrays as arguments, methods may also return arrays.**

A method can return a reference to an array. To do so, the return type of the method must be declared properly. For example, look at the following method definition:

```
public static double[] getArray()
{
   double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };
   return array;
}
```

The `getArray` method returns an array of `double`s. Notice that the return type listed in the method header is `double[]`. The method header is illustrated in Figure 7-13. It indicates that the method returns a reference to a `double` array.

**Figure 7-13** Array reference return type



Inside the method an array of `double`s is created, initialized with some values, and referenced by the `array` variable. Then the `return` statement returns the `array` variable. By returning the `array` variable, the method is returning a reference to the array. The method's return value can be stored in any compatible reference variable, as demonstrated in Code Listing 7-14.

**Code Listing 7-14** `(ReturnArray.java)`

```java
/**
   This program demonstrates how a reference to an
   array can be returned from a method.
*/

public class ReturnArray
{
   public static void main(String[] args)
   {
      double[] values;

      values = getArray();
      for (double num : values)
         System.out.print(num + " ");
   }

   /**
      getArray method
      @return A reference to an array of doubles.
   */

   public static double[] getArray()
   {
      double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };

      return array;
```

```
    }
}
```

**Program Output**

```
1.2  2.3  4.5  6.7  8.9
```

The following statement, which appears in line 12, assigns the array returned by the `getArray` method to the array variable `values`:

```
values = getArray();
```

Then the `for` loop in lines 13 and 14 displays the value of each element of the `values` array.

## 7.6 String Arrays

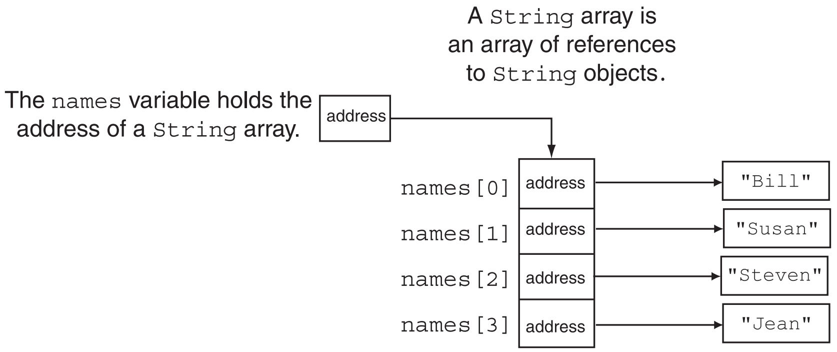
**CONCEPT: An array of `String` objects may be created, but if the array is uninitialized, each `String` in the array must be created individually.**

Java also allows you to create arrays of `String` objects. Here is a statement that creates an array of `String` objects initialized with values:

```
String[] names = { "Bill", "Susan", "Steven", "Jean" };
```

In memory, an array of `String` objects is arranged differently than an array of a primitive data type. In order to use a `String` object, you must have a reference to the `String` object. So, an array of `String` objects is really an array of references to `String` objects. Figure 7-14 illustrates how the `names` variable will reference an array of references to `String` objects.

**Figure 7-14** The names variable references a `String` array


A String array is an array of references to String objects.
The names variable holds the address of a String array.
address
names[0]
names[1]
names[2]
names[3]
address
address
address
address
"Bill"
"Susan"
"Steven"
"Jean"

Each element in the `names` array is a reference to a `String` object. The `names[0]` element references a `String` object containing “Bill”, the `names[1]` element references a `String` object containing “Susan”, and so forth. The program in Code Listing 7-15 demonstrates an array of `String` objects.

**Code Listing 7-15** (MonthDays.java)

```
/**
   This program demonstrates an array of String objects.
*/

public class MonthDays
{
   public static void main(String[] args)
   {
      String[] months = { "January", "February", "March",
                          "April", "May", "June", "July",
                          "August", "September", "October",
                          "November", "December" };

      int[] days = { 31, 28, 31, 30, 31, 30, 31,
                     31, 30, 31, 30, 31 };

       for (int index = 0; index < months.length; index++)
       {
          System.out.println(months[index] + " has " +
                             days[index] + " days.");
       }
   }
}
```

**Program Output**

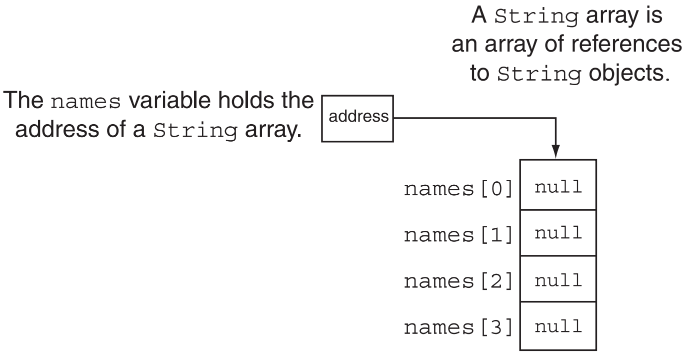
```
January has 31 days.
February has 28 days.
March has 31 days.
April has 30 days.
May has 31 days.
June has 30 days.
July has 31 days.
August has 31 days.
September has 30 days.
October has 31 days.
November has 30 days.
December has 31 days.
```

As with the primitive data types, an initialization list automatically causes an array of `String` objects to be created in memory. If you do not provide an initialization list, you must use the `new` key word to create the array. Here is an example:

```
final int SIZE = 4;
String[] names = new String[SIZE];
```

This statement creates an array of four references to `String` objects, as shown in Figure 7-15. Notice that the array is an array of four uninitialized `String` references. Because they do not reference any objects, they are set to `null`.

**Figure 7-15** An uninitialized String array


A String array is an array of references to String objects.
The names variable holds the address of a String array.
address
names[0]
null
names[1]
null
names[2]
null
names[3]
null

When you create an uninitialized array of `String` objects, you must assign a value to each element in the array that you intend to use. Here is an example:

```
final int SIZE = 4;
String[] names = new String[SIZE];
names[0] = "Bill";
names[1] = "Susan";
names[2] = "Steven";
names[3] = "Jean";
```

After these statements execute, each element of the `names` array will reference a `String` object.

## Calling `String` Methods from an Array Element

Recall from Chapter 2 that `String` objects have several methods. For example, the `toUpperCase` method returns the uppercase equivalent of a `String` object. Because each element of a `String` array is a `String` object, you can use an element to call a `String` method. For example, the following statement uses element 0 of the `names` array to call the `toUpperCase` method:

```
System.out.println(names[0].toUpperCase());
```

The following code shows another example. It uses element 3 of the `names` array to call the `charAt` method. When this code executes, the first character of the string stored in `names[3]` will be assigned to the `letter` variable.

```
// Declare a char variable named letter.
char letter;
// Assign the first character in names[3] to letter.
letter = names[3].charAt(0);
```

**TIP:** Arrays have a field named `length` and `String` objects have a method named `length`. When working with `String` arrays, do not confuse the two. The following loop displays the length of each string held in names, which is assumed to be a `String` array. Note that the loop uses both the array's `length` field and each element's `length` method.

```
for (int i = 0; i < names.length; i++)
    System.out.println(names[i].length());
```

Because the array's `length` member is a field, you do not write a set of parentheses after its name. You do write the parentheses after the name of the `String` class's `length` method.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

7.15 a) Write a statement that declares a `String` array initialized with the following strings: "Mercury", "Venus", "Earth", and "Mars".
  b) Write a loop that displays the contents of each element in the array you declared in A.
  c) Write a loop that displays the first character of the strings stored in each element of the array you declared in A. (*Hint*: Use the `String` class's `charAt` method discussed in Chapter 2.)

## 7.7 Arrays of Objects

**CONCEPT: You may create arrays of objects that are instances of classes that you have written.**

Like any other data type, you can create arrays of class objects. For example, recall the `BankAccount` class that we developed in Chapter 6. An array of `BankAccount` objects could be created to represent all of the bank accounts owned by a single person. The following code declares an array of five `BankAccount` objects:

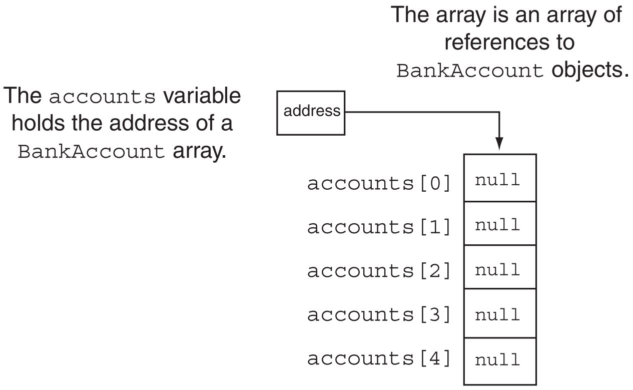
```
final int NUM_ACCOUNTS = 5;
BankAccount[] accounts = new BankAccount[NUM_ACCOUNTS];
```

The variable that references the array is named `accounts`. As with `String` arrays, each element in this array is a reference variable, as illustrated in Figure 7-16.

Notice from the figure that each element of the array is initialized with the value `null`. This is a special value in Java that indicates the array elements do not yet reference objects. You must individually create the objects that each element will reference. The following code uses a loop to create objects for each element:

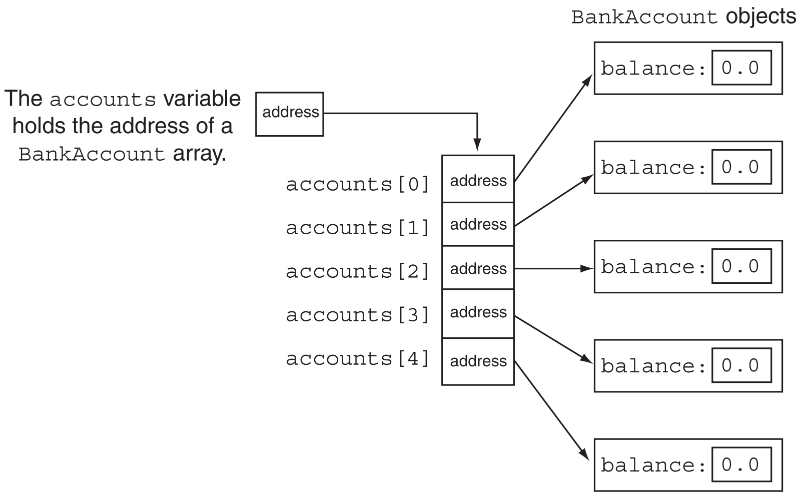
```
for (int index = 0; index < accounts.length; index++)
    accounts[index] = new BankAccount();
```

**Figure 7-16** The accounts variable references an array of references


The array is an array of references to BankAccount objects.
The accounts variable holds the address of a BankAccount array.
address
accounts[0]
accounts[1]
accounts[2]
accounts[3]
accounts[4]
null
null
null
null
null

In this code, the no-arg constructor is called for each object. Recall that the `BankAccount` class has a no-arg constructor that assigns 0.0 to the `balance` field. After the loop executes, each element of the `accounts` array will reference a `BankAccount` object, as shown in Figure 7-17.

**Figure 7-17** Each element of the array references an object


BankAccount objects
The accounts variable holds the address of a BankAccount array.
address
accounts[0]
accounts[1]
accounts[2]
accounts[3]
accounts[4]
address
address
address
address
address
balance: 0.0
balance: 0.0
balance: 0.0
balance: 0.0
balance: 0.0

Objects in an array are accessed with subscripts, just like any other data type in an array. For example, the following code uses the `accounts[2]` element to call the `setBalance` and `withdraw` methods:

```
accounts[2].setBalance(2500.0);
accounts[2].withdraw(500.0);
```

Code Listing 7-16 shows a complete program that uses an array of objects.

**Code Listing 7-16** `(ObjectArray.java)`

```
import java.util.Scanner;

/**
   This program works with an array of three
   BankAccount objects.
*/

public class ObjectArray
{
   public static void main(String[] args)
   {
      final int NUM_ACCOUNTS = 3; // Number of accounts

      // Create an array that can reference
      // BankAccount objects.
      BankAccount[] accounts = new BankAccount[NUM_ACCOUNTS];

      // Create objects for the array.
      createAccounts(accounts);

      // Display the balances of each account.
      System.out.println("Here are the balances " +
                         "for each account:");

      for (int index = 0; index < accounts.length; index++)
      {
         System.out.println("Account " + (index + 1) +
                    ": $" + accounts[index].getBalance());
      }
   }

   /**
      The createAccounts method creates a BankAccount
      object for each element of an array. The user
      is asked for each account's balance.
      @param array The array to reference the accounts
   */

   private static void createAccounts(BankAccount[] array)
   {
      double balance;    // To hold an account balance

      // Create a Scanner object.
      Scanner keyboard = new Scanner(System.in);

```

```
         // Create the accounts.
         for (int index = 0; index < array.length; index++)
         {
            // Get the account's balance.
            System.out.print("Enter the balance for " +
                             "account " + (index + 1) + ": ");
            balance = keyboard.nextDouble();

            // Create the account.
            array[index] = new BankAccount(balance);
          }
      }
   }
```

**Program Output with Example Input Shown in Bold**

```
Enter the balance for account 1: **2500.0 [Enter]**
Enter the balance for account 2: **5000.0 [Enter]**
Enter the balance for account 3: **1500.0 [Enter]**
Here are the balances for each account:
Account 1: $2500.0
Account 2: $5000.0
Account 3: $1500.0
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

7.16 Recall that we discussed a `Rectangle` class in Chapter 6. Write code that declares a `Rectangle` array with five elements. Instantiate each element with a `Rectangle` object. Use the `Rectangle` constructor to initialize each object with values for the `length` and `width` fields.

## 7.8 The Sequential Search Algorithm

**CONCEPT: A search algorithm is a method of locating a specific item in a larger collection of data. This section discusses the sequential search algorithm, which is a simple technique for searching the contents of an array.**

It is very common for programs not only to store and process information stored in arrays, but also to search arrays for specific items. This section shows you how to use the simplest of all search algorithms—the sequential search.

The sequential search algorithm uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for and stops when the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm unsuccessfully searches to the end of the array.

The `SearchArray` program shown in Code Listing 7-17 searches the five-element array `tests` to find a score of 100. It uses a method, `sequentialSearch`, to find the value in the array. The array that is passed as an argument into the `array` parameter is searched for an occurrence of the number passed into `value`. If the number is found, its array subscript is returned. Otherwise, −1 is returned, indicating the value did not appear in the array.

**Code Listing 7-17** (`SearchArray.java`)

```
/**
   This program sequentially searches an
   int array for a specified value.
*/

public class SearchArray
{
   public static void main(String[] args)
   {
      int[] tests = { 87, 75, 98, 100, 82 };
      int results;

      // Search the array for the value 100.
      results = sequentialSearch(tests, 100);

      // Determine whether 100 was found and
      // display an appropriate message.
      if (results == -1)
      {
          System.out.println("You did not " +
                        "earn 100 on any test.");
      }
      else
      {
          System.out.println("You earned 100 " +
                        "on test " + (results + 1));
      }
   }

   /**
      The sequentialSearch method searches an array for
      a value.
      @param array The array to search.
      @param value The value to search for.
      @return The subscript of the value if found in the
              array, otherwise -1.
   */

   public static int sequentialSearch(int[] array,
                                      int value)
   {
       int index;         // Loop control variable
       int element;       // Element the value is found at
       boolean found;     // Flag indicating search results

       // Element 0 is the starting point of the search.
       index = 0;

       // Store the default values element and found.
       element = -1;
       found = false;

       // Search the array.
       while (!found && index < array.length)
       {
          if (array[index] == value)
          {
             found = true;
             element = index;
          }
          index++;
       }

       return element;
   }
}
```

**Program Output**

```
You earned 100 on test 4
```

**NOTE:** The reason −1 is returned when the search value is not found in the array is because −1 is not a valid subscript.

See the `PinTester` Class Case Study on this book's companion Web site (available at www.pearsonhighered.com/gaddis) for another example using arrays. Also, see the companion Web site for the Bonus Section on Parallel Arrays to learn about another programming technique using arrays.

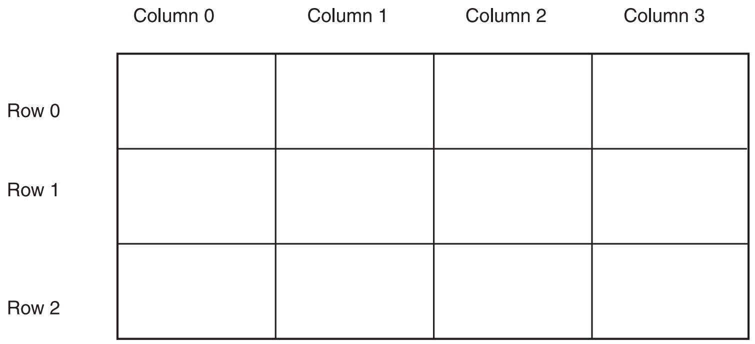
## 7.9 Two-Dimensional Arrays

**CONCEPT: A two-dimensional array is an array of arrays. It can be thought of as having rows and columns.**

An array is useful for storing and working with a set of data. Sometimes, though, it's necessary to work with multiple sets of data. For example, in a grade-averaging program a teacher might record all of one student's test scores in an array of `double`s. If the teacher has 30 students, that means she'll need 30 arrays to record the scores for the entire class. Instead of defining 30 individual arrays, however, it would be better to define a two-dimensional array.

The arrays that you have studied so far are one-dimensional arrays. They are called *one-dimensional* because they can hold only one set of data. Two-dimensional arrays, which are sometimes called *2D arrays*, can hold multiple sets of data. Although a two-dimensional array is actually an array of arrays, it's best to think of it as having rows and columns of elements, as shown in Figure 7-18. This figure shows an array of test scores, having three rows and four columns.

**Figure 7-18** Rows and columns



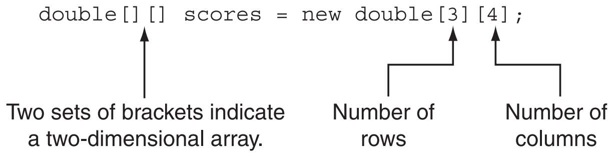
The array shown in the figure has three rows (numbered 0 through 2) and four columns (numbered 0 through 3). There are a total of 12 elements in the array.

To declare a two-dimensional array, two sets of brackets and two size declarators are required: The first one is for the number of rows and the second one is for the number of columns. Here is an example declaration of a two-dimensional array with three rows and four columns:

```
double[][] scores = new double[3][4];
```

The two sets of brackets in the data type indicate that the `scores` variable will reference a two-dimensional array. The numbers `3` and `4` are size declarators. The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice that each size declarator is enclosed in its own set of brackets. This is illustrated in Figure 7-19.

**Figure 7-19** Declaration of a two-dimensional array



```
double[][] scores = new double[3][4];
```

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `scores` array, the elements in row 0 are referenced as follows:

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

The elements in row 1 are as follows:

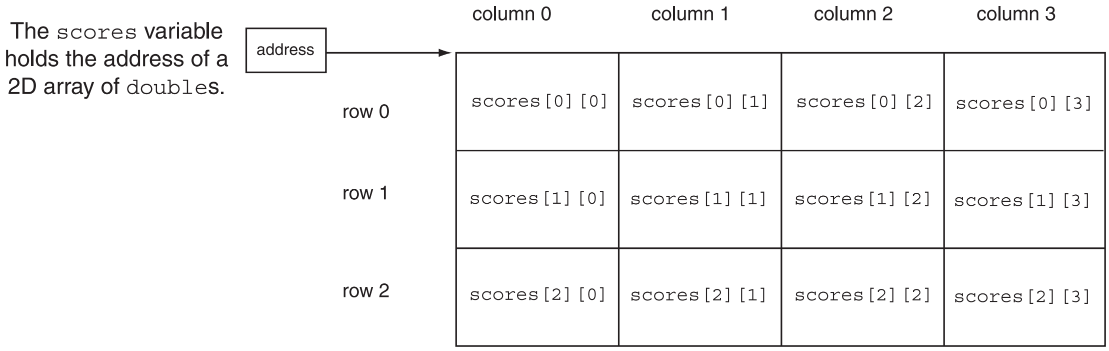
```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

And the elements in row 2 are as follows:

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

Figure 7-20 illustrates the array with the subscripts shown for each element.

**Figure 7-20** Subscripts for each element of the `scores` array



The `scores` variable holds the address of a 2D array of `double`s.

| | column 0 | column 1 | column 2 | column 3 |
|---|---|---|---|---|
| row 0 | `scores[0][0]` | `scores[0][1]` | `scores[0][2]` | `scores[0][3]` |
| row 1 | `scores[1][0]` | `scores[1][1]` | `scores[1][2]` | `scores[1][3]` |
| row 2 | `scores[2][0]` | `scores[2][1]` | `scores[2][2]` | `scores[2][3]` |

To access one of the elements in a two-dimensional array, you must use both subscripts. For example, the following statement stores the number 95 in `scores[2][1]`:

```
scores[2][1] = 95;
```

Programs that process two-dimensional arrays can do so with nested loops. For example, the following code prompts the user to enter a score, once for each element in the array:

```
final int ROWS = 3;
final int COLS = 4;
double[][] scores = new double[ROWS][COLS];
for (int row = 0; row < ROWS; row++)
{
   for (int col = 0; col < COLS; col++)
   {
      System.out.print("Enter a score: ");
      scores[row][col] = keyboard.nextDouble();
   }
}
```

And the following code displays all the elements in the `scores` array:

```
for (int row = 0; row < ROWS; row++)
{
   for (int col = 0; col < COLS; col++)
   {
      System.out.println(scores[row][col]);
   }
}
```

The program in Code Listing 7-18 uses a two-dimensional array to store corporate sales data. The array has three rows (one for each division of the company) and four columns (one for each quarter).

**Code Listing 7-18** **(`CorpSales.java`)**

```
import java.util.Scanner;

/**
   This program demonstrates a two-dimensional array.
*/

public class CorpSales
{
   public static void main(String[] args)
   {
      final int DIVS = 3; // Three divisions in the company
      final int QTRS = 4; // Four quarters
      double totalSales = 0.0; // Accumulator

      // Create an array to hold the sales for each
      // division, for each quarter.
      double[][] sales = new double[DIVS][QTRS];

       // Create a Scanner object for keyboard input.
       Scanner keyboard = new Scanner(System.in);

       // Display an introduction.
       System.out.println("This program will calculate the " +
                          "total sales of");
       System.out.println("all the company's divisions. " +
                          "Enter the following sales data:");

       // Nested loops to fill the array with quarterly
       // sales figures for each division.
       for (int div = 0; div < DIVS; div++)
       {
           for (int qtr = 0; qtr < QTRS; qtr++)
           {
              System.out.printf("Division %d, Quarter %d: $",
                                (div + 1), (qtr + 1));
              sales[div][qtr] = keyboard.nextDouble();
           }
           System.out.println();  // Print blank line.
       }

       // Nested loops to add all the elements of the array.
       for (int div = 0; div < DIVS; div++)
       {
          for (int qtr = 0; qtr < QTRS; qtr++)
          {
             totalSales += sales[div][qtr];
          }
       }

       // Display the total sales.
       System.out.printf("Total company sales: $%,.2f\n",
                         totalSales);
   }
}
```

**Program Output with Example Input Shown in Bold**

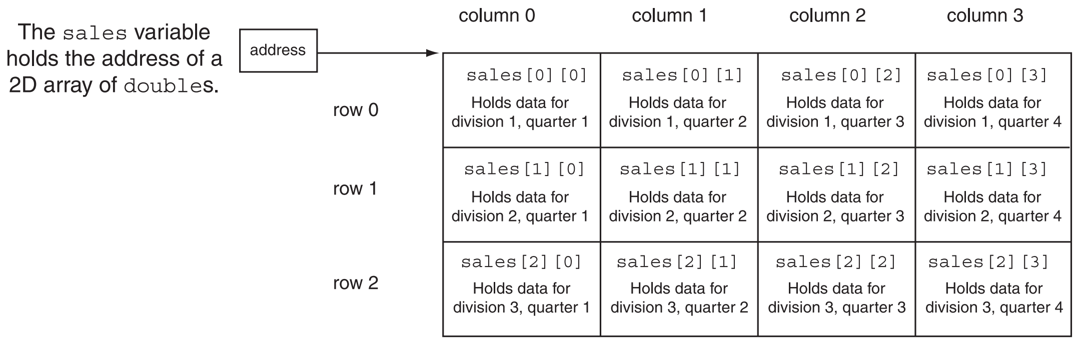
```
This program will calculate the total sales of
all the company's divisions. Enter the following sales data:
Division 1, Quarter 1: $35698.77 [Enter]
Division 1, Quarter 2: $36148.63 [Enter]
Division 1, Quarter 3: $31258.95 [Enter]
Division 1, Quarter 4: $30864.12 [Enter]
Division 2, Quarter 1: $41289.64 [Enter]
Division 2, Quarter 2: $43278.52 [Enter]
Division 2, Quarter 3: $40928.18 [Enter]
Division 2, Quarter 4: $42818.98 [Enter]

Division 3, Quarter 1: $28914.56 [Enter]
Division 3, Quarter 2: $27631.52 [Enter]
Division 3, Quarter 3: $30596.64 [Enter]
Division 3, Quarter 4: $29834.21 [Enter]

Total company sales: $419,262.72
```

Look at the following array declaration in line 17:

```
double[][] sales = new double[DIVS][QTRS];
```

As mentioned earlier, the array has three rows (one for each division) and four columns (one for each quarter) to store the company's sales data. The row subscripts are 0, 1, and 2, and the column subscripts are 0, 1, 2, and 3. Figure 7-21 illustrates how the quarterly sales data is stored in the array.

**Figure 7-21** Division and quarter data stored in the `sales` array


The sales variable holds the address of a 2D array of doubles.
address
column 0
column 1
column 2
column 3
row 0
row 1
row 2
sales[0][0] Holds data for division 1, quarter 1
sales[0][1] Holds data for division 1, quarter 2
sales[0][2] Holds data for division 1, quarter 3
sales[0][3] Holds data for division 1, quarter 4
sales[1][0] Holds data for division 2, quarter 1
sales[1][1] Holds data for division 2, quarter 2
sales[1][2] Holds data for division 2, quarter 3
sales[1][3] Holds data for division 2, quarter 4
sales[2][0] Holds data for division 3, quarter 1
sales[2][1] Holds data for division 3, quarter 2
sales[2][2] Holds data for division 3, quarter 3
sales[2][3] Holds data for division 3, quarter 4

## Initializing a Two-Dimensional Array

When initializing a two-dimensional array, you enclose each row's initialization list in its own set of braces. Here is an example:

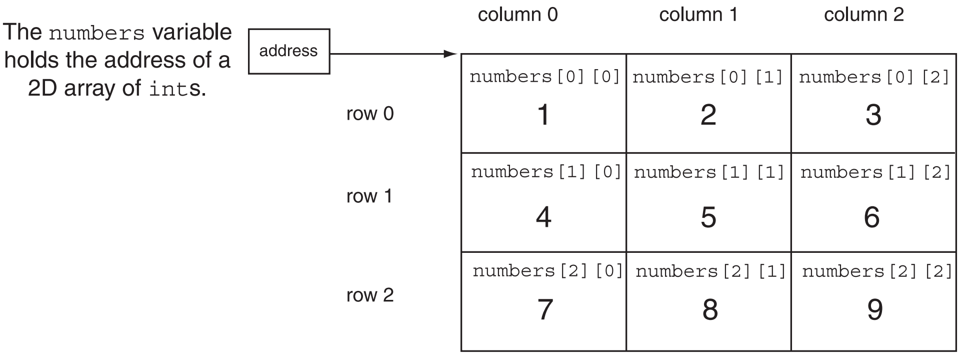
```
int[][] numbers = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

As with one-dimensional arrays, you do not use the `new` key word when you provide an initialization list. Java automatically creates the array and fills its elements with the initialization values. In this example, the initialization values for row 0 are {1, 2, 3}, the initialization values for row 1 are {4, 5, 6}, and the initialization values for row 2 are {7, 8, 9}. So, this statement declares an array with three rows and three columns. For more clarity, the same statement could also be written as follows:

```
int[][] numbers = { {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9} };
```

In either case, the values are assigned to the `numbers` array as illustrated in Figure 7-22.

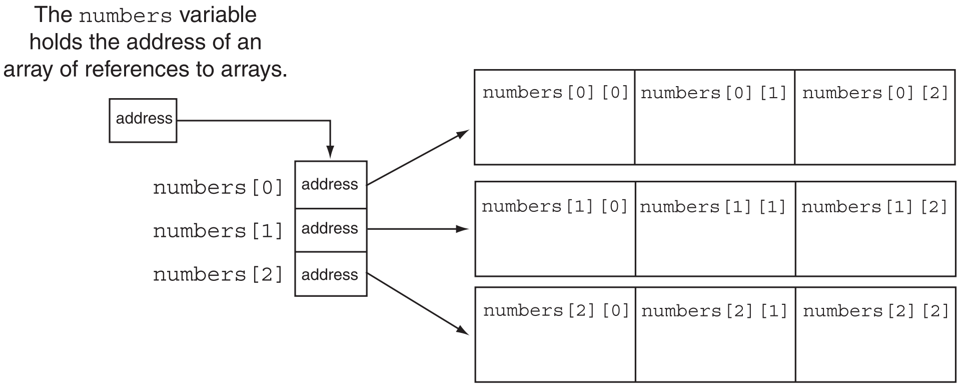
**Figure 7-22** The `numbers` array


The numbers variable holds the address of a 2D array of ints.
address
column 0
column 1
column 2
row 0
row 1
row 2
numbers[0][0]
1
numbers[0][1]
2
numbers[0][2]
3
numbers[1][0]
4
numbers[1][1]
5
numbers[1][2]
6
numbers[2][0]
7
numbers[2][1]
8
numbers[2][2]
9

## The `length` Field in a Two-Dimensional Array

A one-dimensional array has a `length` field that holds the number of elements in the array. A two-dimensional array, however, has multiple `length` fields. It has a `length` field that holds the number of rows, and then each row has a `length` field that holds the number of columns. This makes sense when you think of a two-dimensional array as an array of one-dimensional arrays. Figure 7-22 shows the `numbers` array depicted in rows and columns. Figure 7-23 shows another way of thinking of the `numbers` array: as an array of arrays.

**Figure 7-23** The `numbers` array is an array of arrays


The numbers variable holds the address of an array of references to arrays.
address
numbers[0]
numbers[1]
numbers[2]
address
address
address
numbers[0][0]
numbers[0][1]
numbers[0][2]
numbers[1][0]
numbers[1][1]
numbers[1][2]
numbers[2][0]
numbers[2][1]
numbers[2][2]

As you can see from the figure, the `numbers` variable references a one-dimensional array with three elements. Each of the three elements is a reference to another one-dimensional array. The elements in the array referenced by `numbers[0]` are `numbers[0][0]`, `numbers[0][1]`, and `numbers[0][2]`. This pattern continues with `numbers[1]` and `numbers[2]`. The figure shows a total of four arrays. Each of the arrays in the figure has its own `length` field. The program in Code Listing 7-19 uses these `length` fields to display the number of rows and columns in a two-dimensional array.

**Code Listing 7-19** `(Lengths.java)`

```
/**
   This program uses the length fields of a 2D array
   to display the number of rows, and the number of
   columns in each row.
*/

public class Lengths
{
   public static void main(String[] args)
   {
      // Declare a 2D array with 3 rows
      // and 4 columns.

      int[][] numbers = { { 1,  2, 3,   4 },
                          { 5,  6, 7,   8 },
                          { 9, 10, 11, 12 } };

       // Display the number of rows.
       System.out.println("The number of " +
                  "rows is " + numbers.length);

       // Display the number of columns in each row.
       for (int index = 0; index < numbers.length; index++)
       {
           System.out.println("The number of " +
                 "columns in row " + index + " is " +
                 numbers[index].length);
       }
   }
}
```

**Program Output**

```
The number of rows is 3
The number of columns in row 0 is 4
The number of columns in row 1 is 4
The number of columns in row 2 is 4
```

## Displaying All the Elements of a Two-Dimensional Array

As you have seen in previous example programs, a pair of nested loops can be used to display all the elements of a two-dimensional array. For example, the following code creates the `numbers` array with three rows and four columns, and then displays all the elements in the array:

```
int[][] numbers = { { 1,  2,  3,  4 },
                    { 5,  6,  7,  8 },
                    { 9, 10, 11, 12 } };

for (int row = 0; row < 3; row++)
{
    for (int col = 0; col < 4; col++)
       System.out.println(numbers[row][col]);
}
```

Although this code will display all of the elements, it is limited in the following way: The loops are specifically written to display an array with three rows and four columns. A better approach is to use the array's `length` fields for the upper limit of the subscripts in the loop test expressions. Here are the modified loops:

```
for (int row = 0; row < numbers.length; row++)
{
   for (int col = 0; col < numbers[row].length; col++)
      System.out.println(numbers[row][col]);
}
```

Let's take a closer look at the header for the outer loop:

```
for (int row = 0; row < numbers.length; row++)
```

This loop controls the subscript for the `number` array's rows. Because `numbers.length` holds the number of rows in the array, we have used it as the upper limit for the row subscripts. Here is the header for the inner loop:

```
for (int col = 0; col < numbers[row].length; col++)
```

This loop controls the subscript for the `number` array's columns. Because each row's `length` field holds the number of columns in the row, we have used it as the upper limit for the column subscripts. By using the `length` fields in algorithms that process two-dimensional arrays, you can write code that works with arrays of any number of rows and columns.

## Summing All the Elements of a Two-Dimensional Array

To sum all the elements of a two-dimensional array, you can use a pair of nested loops to add the contents of each element to an accumulator. The following code shows an example:

```
int[][] numbers = { { 1,  2,  3,  4 },
                    { 5,  6,  7,  8 },
                    { 9, 10, 11, 12 } };
```

```
int total = 0;     // Accumulator, set to 0
// Sum the array elements.
for (int row = 0; row < numbers.length; row++)
{
   for (int col = 0; col < numbers[row].length; col++)
      total += numbers[row][col];
}

// Display the sum.
System.out.println("The total is " + total);
```

## Summing the Rows of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each row in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students. Each row in the array is a set of test scores for one student. To get the sum of a student's test scores (perhaps so an average may be calculated), you use a loop to add all the elements in one row. The following code shows an example:

```
int[][] numbers = { { 1,  2,  3,  4 },
                    { 5,  6,  7,  8 },
                    { 9, 10, 11, 12 } };
int total;          // Accumulator
for (int row = 0; row < numbers.length; row++)
{
    // Set the accumulator to 0.
    total = 0;

    // Sum a row.
    for (int col = 0; col < numbers[row].length; col++)
       total += numbers[row][col];

    // Display the row's total.
    System.out.println("Total of row " + row +
                       " is " + total);
}
```

Notice that the `total` variable, which is used as an accumulator, is set to zero just before the inner loop executes, because the inner loop sums the elements of a row and stores the sum in `total`. Therefore, the `total` variable must be set to zero before each iteration of the inner loop.

## Summing the Columns of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each column in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students, and you wish to calculate the class average for each of the test scores. To do this, you calculate the average of each column in the array. This is accomplished with a set of nested loops. The outer loop controls the column subscript and the inner loop controls the row subscript. The inner loop calculates the sum of a column, which is stored in an accumulator. The following code demonstrates:

```
int[][] numbers = { { 1,  2,  3,  4 },
                    { 5,  6,  7,  8 },
                    { 9, 10, 11, 12 } };
int total;  // Accumulator

for (int col = 0; col < numbers[0].length; col++)
{
   // Set the accumulator to 0.
   total = 0;

   // Sum a column.
   for (int row = 0; row < numbers.length; row++)
      total += numbers[row][col];

   // Display the columns's total.
   System.out.println("Total of column " + col +
                      " is " + total);
}
```

## Passing Two-Dimensional Arrays to Methods

When a two-dimensional array is passed to a method, the parameter must be declared as a reference to a two-dimensional array. The following method header shows an example:

```
private static void showArray(int[][] array)
```

This method's parameter, `array`, is declared as a reference to a two-dimensional `int` array. Any two-dimensional `int` array can be passed as an argument to the method. Code Listing 7-20 demonstrates two such methods.

**Code Listing 7-20** (Pass2Darray.java)

```
/**
   This program demonstrates methods that accept
   a two-dimensional array as an argument.
 */

public class Pass2Darray
{
   public static void main(String[] args)
   {
      int[][] numbers = { { 1,  2,  3,  4 },
                          { 5,  6,  7,  8 },
                          { 9, 10, 11, 12 } };

```

```
       // Display the contents of the array.
       System.out.println("Here are the values " +
                          " in the array.");
       showArray(numbers);

       // Display the sum of the array's values.
       System.out.println("The sum of the values " +
                          "is " + arraySum(numbers));
   }

   /**
      The showArray method displays the contents
      of a two-dimensional int array.
      @param array The array to display.
   */

   private static void showArray(int[][] array)
   {
      for (int row = 0; row < array.length; row++)
      {
         for (int col = 0; col < array[row].length; col++)
            System.out.print(array[row][col] + " ");
         System.out.println();
       }
   }

   /**
      The arraySum method returns the sum of the
      values in a two-dimensional int array.
      @param array The array to sum.
      @return The sum of the array elements.
   */

   private static int arraySum(int[][] array)
   {
      int total = 0;    // Accumulator

      for (int row = 0; row < array.length; row++)
      {
         for (int col = 0; col < array[row].length; col++)
            total += array[row][col];
       }

```

```
         return total;
      }
   }
```

**Program Output**

```
Here are the values in the array.
1 2 3 4
5 6 7 8
9 10 11 12
The sum of the values is 78
```

## Ragged Arrays

Because the rows in a two-dimensional array are also arrays, each row can have its own length. When the rows of a two-dimensional array are of different lengths, the array is known as a ragged array. You create a ragged array by first creating a two-dimensional array with a specific number of rows, but no columns. Here is an example:

```
int[][] ragged = new int[4][];
```

This statement partially creates a two-dimensional array. The array can have four rows, but the rows have not yet been created. Next, you create the individual rows as shown in the following code:

```
ragged[0] = new int[3];    // Row 0 has 3 columns.
ragged[1] = new int[4];    // Row 1 has 4 columns.
ragged[2] = new int[5];    // Row 2 has 5 columns.
ragged[3] = new int[6];    // Row 3 has 6 columns.
```

This code creates the four rows. Row 0 has three columns, row 1 has four columns, row 2 has five columns, and row 3 has six columns. The following code displays the number of columns in each row:

```
for (int index = 0; index < ragged.length; index++)
{
   System.out.println("The number of columns " +
                      "in row " + index + " is " +
                      ragged[index].length);
}
```

This code will display the following output:

```
The number of columns in row 0 is 3
The number of columns in row 1 is 4
The number of columns in row 2 is 5
The number of columns in row 3 is 6
```

## 7.10 Arrays with Three or More Dimensions

**CONCEPT: Java does not limit the number of dimensions that an array may have. It is possible to create arrays with multiple dimensions, to model data that occurs in multiple sets.**

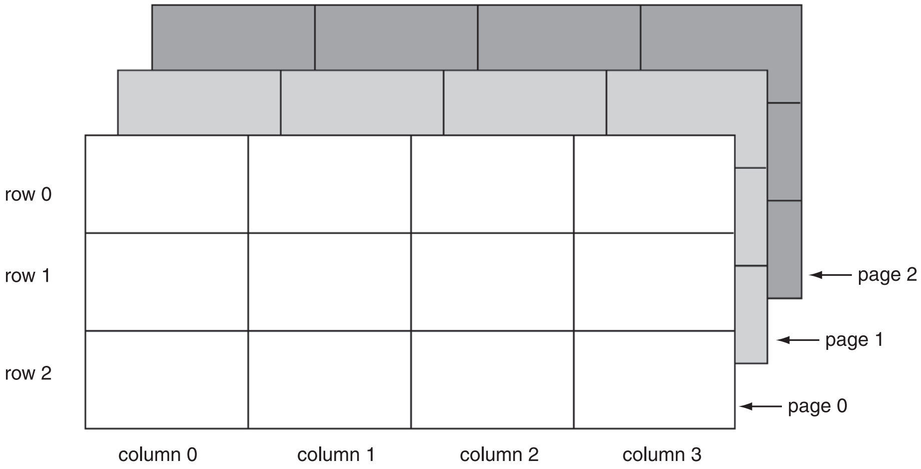
Java allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array declaration:

```
double[][][] seats = new double[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

Figure 7-24 illustrates the concept of a three-dimensional array as "pages" of two-dimensional arrays.

**Figure 7-24** A three-dimensional array



Arrays with more than three dimensions are difficult to visualize, but can be useful in some programming problems. For example, in a factory warehouse where cases of widgets are stacked on pallets, an array with four dimensions could be used to store a part number for each widget. The four subscripts of each element could represent the pallet number, case number, row number, and column number of each widget. Similarly, an array with five dimensions could be used if there were multiple warehouses.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

7.17 A video rental store keeps videos on 50 racks with 10 shelves each. Each shelf holds 25 videos. Declare a three-dimensional array large enough to represent the store's storage system.

# 7.11 The Selection Sort and the Binary Search Algorithms

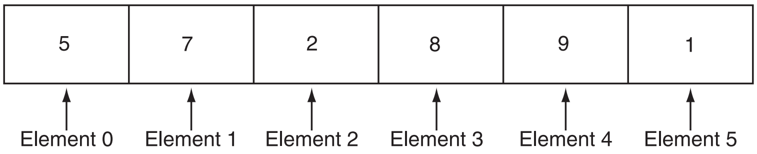
**CONCEPT: A sorting algorithm is used to arrange data into some order. A search algorithm is a method of locating a specific item in a larger collection of data. The selection sort and the binary search are popular sorting and searching algorithms.**

## The Selection Sort Algorithm

Often the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Product codes could be sorted so all the products of the same color are stored together. In this section we explore how to write your own sorting algorithm. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order. The algorithm that we will explore is called the selection sort.

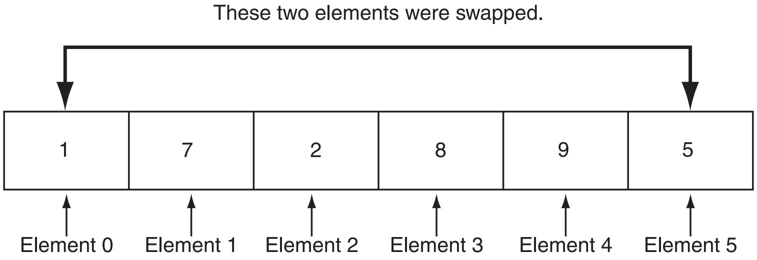
The *selection sort* works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order. Let's see how the selection sort works when arranging the elements of the following array in Figure 7-25.

**Figure 7-25** Values in an array



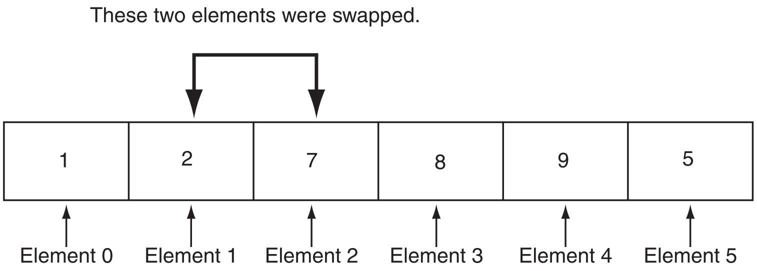
The selection sort scans the array, starting at element 0, and locates the element with the smallest value. The contents of this element are then swapped with the contents of element 0. In this example, the 1 stored in element 5 is swapped with the 5 stored in element 0. After the exchange, the array would appear as shown in Figure 7-26.

**Figure 7-26** Values in array after first swap



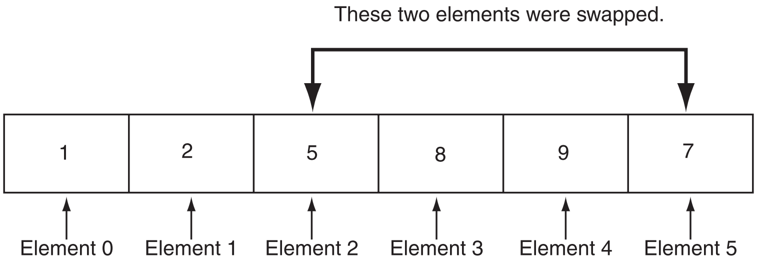
The algorithm then repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. This time, the algorithm begins the scan at element 1. In this example, the contents of element 2 are exchanged with that of element 1. The array would then appear as shown in Figure 7-27.

**Figure 7-27** Values in array after second swap


These two elements were swapped.
1
2
7
8
9
5
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

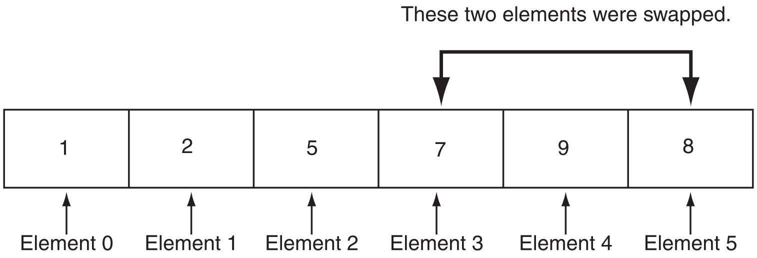
Once again the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value. This element's value is swapped with that of element 2, causing the array to appear as shown in Figure 7-28.

**Figure 7-28** Values in array after third swap


These two elements were swapped.
1
2
5
8
9
7
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

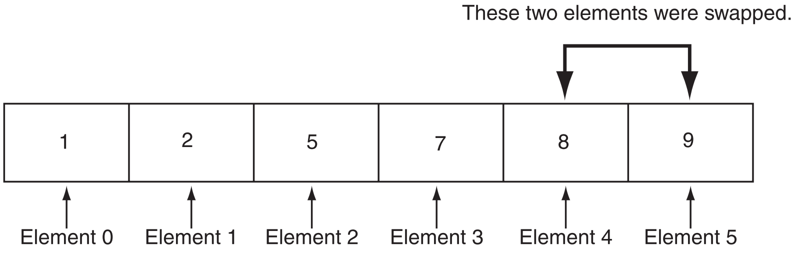
Next, the scanning begins at element 3. Its value is swapped with that of element 5, causing the array to appear as shown in Figure 7-29.

**Figure 7-29** Values in array after fourth swap


These two elements were swapped.
1
2
5
7
9
8
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

At this point there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement as shown in Figure 7-30.

**Figure 7-30** Values in array after fifth swap



Here is the selection sort algorithm in pseudocode:

```
For startScan is each subscript in array from 0 through the next-to-last subscript
   Set minIndex variable to startScan
   Set minValue variable to array[startScan]
   For index is each subscript in array from (startScan + 1) through the last subscript
      If array[index] is less than minValue
         Set minValue to array[index]
         Set minIndex to index
      End If
      Increment index
   End For.
   Set array[minIndex] to array[startScan]
   Set array[startScan] to minValue
End For
```

The following method performs a selection sort on an integer array. The array that is passed as an argument is sorted in ascending order.

```
public static void selectionSort(int[] array)
{
  int startScan, index, minIndex, minValue;

  for (startScan = 0; startScan < (array.length-1); startScan++)
  {
     minIndex = startScan;
     minValue = array[startScan];
     for(index = startScan + 1; index < array.length; index++)
     {
        if (array[index] < minValue)
        {
          minValue = array[index];
```

```
               minIndex = index;
            }
         }
         array[minIndex] = array[startScan];
         array[startScan] = minValue;
      }
   }
```

The *SelectionSortDemo.java* program demonstrates the `selectionSort` method. You can download this chapter's source code from the book's companion Web site at www.pearsonhighered.com/gaddis.

## The Binary Search Algorithm

This chapter previously presented the sequential search algorithm for searching an array. The advantage of the sequential search is its simplicity. It is easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements in order to find a value stored in the last element. In an average case, an item is just as likely to be found near the end of the array as near the beginning. Typically, for an array of N items, the sequential search will locate an item in N/2 attempts. If an array has 50,000 elements, the sequential search will make a comparison with 25,000 of them in a typical case.

This is assuming, of course, that the search item is consistently found in the array. (N/2 is the average number of comparisons. The maximum number of comparisons is always N.) When the sequential search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Obviously, the sequential search should not be used on large arrays if speed is important.

The binary search is a clever algorithm that is much more efficient than the sequential search. Its only requirement is that the values in the array must be sorted in ascending order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm tests its middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found, or there are no more elements to test. Here is the pseudocode for a method that performs a binary search on an array:

```
Set first to 0
Set last to the last subscript in the array
Set position to - 1
Set found to false
While found is not true and first is less than or equal to last
   Set middle to the subscript half way between array[first]and array[last]
   If array[middle] equals the desired value
      Set found to true
      Set position to middle
   Else If array[middle] is greater than the desired value
      Set last to middle - 1
   Else
      Set first to middle + 1
   End If
End While
Return position
```

This algorithm uses three variables to mark positions within the array: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's `first` and `last` elements. The subscript of the element halfway between `first` and `last` is calculated and stored in the `middle` variable. If the element in the middle of the array does not contain the search value, the `first` or `last` variables are adjusted so that only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The following method performs a binary search on an integer array. The first parameter, `array`, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, −1 is returned, indicating the value did not appear in the array.

```
public static int binarySearch(int[] array, int value)
{
   int first;          // First array element
   int last;           // Last array element
   int middle;         // Midpoint of search
   int position;       // Position of search value
   boolean found;      // Flag
   // Set the inital values.
   first = 0;
   last = array.length - 1;
   position = -1;
   found = false;
   // Search for the value.
   while (!found && first <= last)
   {
         // Calculate midpoint
         middle = (first + last) / 2;
         // If value is found at midpoint...
         if (array[middle] == value)
         {
           found = true;
           position = middle;
         }
         // else if value is in lower half...
         else if (array[middle] > value)
            last = middle - 1;
         // else if value is in upper half....
         else
           first = middle + 1;
        }
        // Return the position of the item, or -1
        // if it was not found.
        return position;
      }
```

The `BinarySearchDemo` program demonstrates this method. You can download this chapter's source code from the book's companion Web site at www.pearsonhighered.com/gaddis.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

7.18 What value in an array does the selection sort algorithm look for first?

When the selection sort finds this value, what does it do with it?

7.19 How many times will the selection sort swap the smallest value in an array with another value?

7.20 Describe the difference between the sequential search and the binary search.

7.21 On average, with an array of 20,000 elements, how many comparisons will the sequential search perform? (Assume the items being searched for are consistently found in the array.)

7.22 If a sequential search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

## 7.12 Command-Line Arguments and Variable-Length Argument Lists

**CONCEPT: When you invoke a Java program from the operating system command line, you can specify arguments that are passed into the `main` method of the program. In addition, you can write a method that takes a variable number of arguments. When the method runs, it can determine the number of arguments that were passed to it and act accordingly.**

## Command-Line Arguments

Every program you have seen in this book and every program you have written uses a static `main` method with a header that looks like this:

```
public static void main(String[] args)
```

Inside the parentheses of the method header is the declaration of a parameter named `args`. This parameter is an array name. As its declaration indicates, it is used to reference an array of `Strings`. The array that is passed into the `args` parameter comes from the operating system command line. For example, look at Code Listing 7-21.

**Code Listing 7-21** (`CommandLine.java`)

```
/**
   This program displays the arguments passed to
   it from the operating system command line.
*/

public class CommandLine
{
   public static void main(String[] args)
   {
      for (int index = 0; index < args.length; index++)
         System.out.println(args[index]);
   }
}
```

If this program is compiled and then executed with the following command:

```
java CommandLine How does this work?
```

its output will be as follows:

```
How
does
this
work?
```

Any items typed on the command line, separated by spaces, and after the name of the class are considered to be one or more arguments that are to be passed into the `main` method. In the previous example, four arguments are passed into `args`. The word “How” is passed into `args[0]`, “does” is passed into `args[1]`, “this” is passed into `args[2]`, and “work?” is passed into `args[3]`. The `for` loop in `main` simply displays each argument.

**NOTE:** It is not required that the name of `main`’s parameter array be `args`. You can name it anything you wish. It is a standard convention, however, for the name `args` to be used.

## Variable-Length Argument Lists

Java provides a mechanism known as *variable-length argument lists,* which makes it possible to write a method that takes a variable number of arguments. In other words, you can write a method that accepts any number of arguments when it is called. When the method runs, it can determine the number of arguments that were passed to it and act accordingly.

For example, suppose we need to write a method named `sum` that can accept any number of `int` values and then return the sum of those values. We might call the method as shown here:

```
result = sum(10, 20);
```

Here we pass two arguments to the method: 10 and 20. After this code executes, the value 30 would be stored in the `result` variable. But, the method does not have to accept two arguments each time it is called. We could call the method again with a different number of arguments, as shown here:

```
int firstVal = 1, secondVal = 2, thirdVal = 3, fourthVal = 4;
result = sum(firstVal, secondVal, thirdVal, fourthVal);
```

Here we pass four arguments to the method: `firstVal` (which is set to 1), `secondVal` (which is set to 2), `thirdVal` (which is set to 3), and `fourthVal` (which is set to 4). After this code executes, the value 10 would be stored in the `result` variable. Here's the code for the `sum` method:

```
public static int sum(int... numbers)
{
   int total = 0; // Accumulator

   // Add all the values in the numbers array.
   for (int val : numbers)
     total += val;

   // Return the total.
   return total;
}
```

Notice the declaration of the `numbers` parameter in the method header. The ellipsis (three periods) that follows the data type indicates that `numbers` is a special type of parameter known as a *vararg parameter*. A vararg parameter can take a variable number of arguments.

In fact, vararg parameters are actually arrays. In the `sum` method, the `numbers` parameter is an array of `int`s. All of the arguments that are passed to the `sum` method are stored in the elements of the `numbers` array. As you can see from the code, the method uses the enhanced `for` loop to step through the elements of the `numbers` array, adding up the values stored in its elements. (The *VarargsDemo1.java* program in this chapter's source code demonstrates the `sum` method.)

You can also write a method to accept a variable number of object references as arguments. For example, the program in Code Listing 7-22 shows a method that accepts a variable number of references to `BankAccount` objects. The method returns the total of the objects' `balance` fields.

**Code Listing 7-22** (`VarargsDemo2.java`)

```java
/**
   This program demonstrates a method that accepts
   a variable number of arguments (varargs).
*/

public class VarargsDemo2
{
   public static void main(String[] args)
   {
      double total; // To hold the total balances

      // Create BankAccount object with $100.
      BankAccount account1 = new BankAccount(100.0);

      // Create BankAccount object with $500.
      BankAccount account2 = new BankAccount(500.0);

      // Create BankAccount object with $1500.
      BankAccount account3 = new BankAccount(1500.0);

      // Call the method with one argument.
      total = totalBalance(account1);
      System.out.println("Total: $" + total);

      // Call the method with two arguments.
      total = totalBalance(account1, account2);
      System.out.println("Total: $" + total);

      // Call the method with three arguments.
      total = totalBalance(account1, account2, account3);
      System.out.println("Total: $" + total);
   }

   /**
      The totalBalance method takes a variable number
      of BankAccount objects and returns the total
      of their balances.
      @param accounts The target account or accounts.
      @return The sum of the account balances
   */

   public static double totalBalance(BankAccount... accounts)
   {
      double total = 0.0; // Accumulator

      // Add all the values in the accounts array.
      for (BankAccount acctObject : accounts)
         total += acctObject.getBalance();

      // Return the total.
      return total;
   }
}
```

**Program Output**

```
Total: $100.0
Total: $600.0
Total: $2100.0
```

You can write a method to accept a mixture of fixed arguments and a variable-length argument list. For example, suppose we want to write a method named `courseAverage`, which accepts the name of a course as a `String`, and a variable-length list of test scores as `double`s. We could write the method header as follows:

```
public static void courseAverage(String course, double... scores)
```

This method has a regular `String` parameter named `course`, and a vararg parameter named `scores`. When we call this method, we always pass a `String` argument, then a list of `double` values. (This method is demonstrated in the program *VarargsDemo3.java*, which is in this chapter's source code folder.) Note that when a method accepts a mixture of fixed arguments and a variable-length argument list, the vararg parameter must be the last one declared.

You can also pass an array to a vararg parameter. This is demonstrated in the program *VarargsDemo4.java*. You can download this chapter's source code from the book's companion Web site at www.pearsonhighered.com/gaddis.

## 7.13 The ArrayList Class

**CONCEPT: `ArrayList` is a class in the Java API that is similar to an array and allows you to store objects. Unlike an array, an `ArrayList` object's size is automatically adjusted to accommodate the number of items being stored in it.**

The Java API provides a class named `ArrayList`, which can be used for storing and retrieving objects. Once you create an `ArrayList` object, you can think of it as a container for holding other objects. An `ArrayList` object is similar to an array of objects, but offers many advantages over an array. Here are a few:

- An `ArrayList` object automatically expands as items are added to it.
- In addition to adding items to an `ArrayList`, you can remove items as well.
- An `ArrayList` object automatically shrinks as items are removed from it.

The `ArrayList` class is in the `java.util` package, so the following `import` statement is required:

```
import java.util.ArrayList;
```

## Creating and Using an `ArrayList` Object

Here is an example of how you create an `ArrayList` object:

```
ArrayList<String> nameList = new ArrayList<String>();
```

This statement creates a new `ArrayList` object and stores its address in the `nameList` variable. Notice that in this example the word `String` is written inside angled brackets `<>` immediately after the word `ArrayList`. This specifies that the `ArrayList` can hold `String` objects. If we try to store any other type of object in this `ArrayList`, an error will occur. (Later in this section, you will see an example that creates an `ArrayList` for holding other types of objects.)

To add items to the `ArrayList` object, you use the add method. For example, the following statements add a series of `String` objects to `nameList`:

```
nameList.add("James");
nameList.add("Catherine");
nameList.add("Bill");
```

After these statements execute, `nameList` will hold three references to `String` objects. The first will reference "James", the second will reference "Catherine", and the third will reference "Bill".

The items that are stored in an `ArrayList` have a corresponding index. The index specifies the item's location in the `ArrayList`, so it is much like an array subscript. The first item that is added to an `ArrayList` is stored at index 0. The next item that is added to the `ArrayList` is stored at index 1, and so forth. After the previously shown statements execute, "James" will be stored at index 0, "Catherine" will be stored at index 1, and "Bill" will be stored at index 2.

The `ArrayList` class has a `size` method that reports the number of items stored in an `ArrayList`. It returns the number of items as an `int`. For example, the following statement uses the method to display the number of items stored in `nameList`:

```
System.out.println("The ArrayList has " +
                   nameList.size() +
                   " objects stored in it.");
```

Assuming that `nameList` holds the `String`s "James", "Catherine", and "Bill", the following statement will display:

```
The ArrayList has 3 objects stored in it.
```

The `ArrayList` class's `get` method returns the item stored at a specific index. You pass the index as an argument to the method. For example, the following statement will display the item stored at index 1 of `nameList`:

```
System.out.println(nameList.get(1));
```

The program in Code Listing 7-23 demonstrates the topics discussed so far.

**Code Listing 7-23** (`ArrayListDemo1.java`)

```
 1  import java.util.ArrayList;
 2  
 3  /**
 4     This program demonstrates an ArrayList.
 5  */
 6  
 7  public class ArrayListDemo1
 8  {
 9     public static void main(String[] args)
10     {
11        // Create an ArrayList to hold some names.
12        ArrayList<String> nameList = new ArrayList<String>();
13  
14        // Add some names to the ArrayList.
15        nameList.add("James");
16        nameList.add("Catherine");
17        nameList.add("Bill");
18  
19        // Display the size of the ArrayList.
20        System.out.println("The ArrayList has " +
21                           nameList.size() +
22                           " objects stored in it.");
23  
24        // Now display the items in nameList.
25        for (int index = 0; index < nameList.size(); index++)
26          System.out.println(nameList.get(index));
27     }
28  }
```

**Program Output**

```
The ArrayList has 3 objects stored in it.
James
Catherine
Bill
```

Notice in line 25 that the `for` loop uses the value returned from `nameList`'s `size` method to control the number of times the loop iterates. This is to prevent a bounds checking error from occurring. The last item stored in an `ArrayList` will have an index that is 1 less than the `size` of the `ArrayList`. If you pass a value larger than this to the `get` method, an error will occur.

## Using the Enhanced `for` Loop with an `ArrayList`

Earlier in this chapter, you saw how the enhanced `for` loop can be used to iterate over each `element` in an array. You can also use the enhanced `for` loop to iterate over each item in an

`ArrayList`. Code Listing 7-24 demonstrates. The enhanced `for` loop is used in lines 26 and 27 to display all of the items stored in the `ArrayList`.

**Code Listing 7-24** (`ArrayListDemo2.java`)

```
 1  import java.util.ArrayList;
 2
 3  /**
 4     This program demonstrates how the enhanced for loop
 5     can be used with an ArrayList.
 6  */
 7
 8  public class ArrayListDemo2
 9  {
10     public static void main(String[] args)
11     {
12        // Create an ArrayList to hold some names.
13        ArrayList<String> nameList = new ArrayList<String>();
14
15        // Add some names to the ArrayList.
16        nameList.add("James");
17        nameList.add("Catherine");
18        nameList.add("Bill");
19
20        // Display the size of the ArrayList.
21        System.out.println("The ArrayList has " +
22                           nameList.size() +
23                           " objects stored in it.");
24
25        // Now display the items in nameList.
26        for (String name : nameList)
27           System.out.println(name);
28     }
29  }
```

**Program Output**

```
The ArrayList has 3 objects stored in it.
James
Catherine
Bill
```

## The `ArrayList` Class's `toString` method

The `ArrayList` class has a `toString` method that returns a string representing all of the items stored in an `ArrayList` object. For example, suppose we have set up the `nameList` object as previously shown, with the `Strings` “James”, “Catherine”, and “Bill”. We could use the following statement to display all of the names:

```
System.out.println(nameList);
```

The contents of the `ArrayList` will be displayed in the following manner:

```
[James, Catherine, Bill]
```

This is demonstrated in the program *ArrayListToString.java*, which is in this chapter’s source code folder, available at www.pearsonhighered.com/gaddis.

## Removing an Item from an `ArrayList`

The `ArrayList` class has a `remove` method that removes an item at a specific index. You pass the index as an argument to the method. The program in Code Listing 7-25 demonstrates.

**Code Listing 7-25** `(ArrayListDemo3.java)`

```
import java.util.ArrayList;

/**
   This program demonstrates an ArrayList.
*/

public class ArrayListDemo3
{
   public static void main(String[] args)
   {
      // Create an ArrayList to hold some names.
      ArrayList<String> nameList = new ArrayList<String>();

      // Add some names to the ArrayList.
      nameList.add("James");
      nameList.add("Catherine");
      nameList.add("Bill");

      // Display the items in nameList and their indices.
      for (int index = 0; index < nameList.size(); index++)
      {
         System.out.println("Index: " + index + " Name: " +
                            nameList.get(index));
      }

      // Now remove the item at index 1.
      nameList.remove(1);

      System.out.println("The item at index 1 is removed. " +
                         "Here are the items now.");

      // Display the items in nameList and their indices.
33     for (int index = 0; index < nameList.size(); index++)
34     {
35        System.out.println("Index: " + index + " Name: " +
36                           nameList.get(index));
37      }
38    }
39 }
```

**Program Output**

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
The item at index 1 is removed. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Bill
```

Note that when the item at index 1 was removed (in line 27), the item that was previously stored at index 2 was shifted in position to index 1. When an item is removed from an `ArrayList`, the items that come after it are shifted downward in position to fill the empty space. This means that the index of each item after the removed item will be decreased by one.

Note that an error will occur if you call the `remove` method with an invalid index.

## Inserting an Item

The `add` method, as previously shown, adds an item at the last position in an `ArrayList` object. The `ArrayList` class has an overloaded version of the `add` method that allows you to add an item at a specific index. This causes the item to be inserted into the `ArrayList` object at a specific position. The program in Code Listing 7-26 demonstrates.

**Code Listing 7-26** (`ArrayListDemo4.java`)

```
import java.util.ArrayList;

/**
   This program demonstrates inserting an item.
*/

public class ArrayListDemo4
{
   public static void main(String[] args)
   {
     // Create an ArrayList to hold some names.
     ArrayList<String> nameList = new ArrayList<String>();

     // Add some names to the ArrayList.
     nameList.add("James");
     nameList.add("Catherine");
17        nameList.add("Bill");
18
19        // Display the items in nameList and their indices.
20        for (int index = 0; index < nameList.size(); index++)
21        {
22          System.out.println("Index: " + index + " Name: " +
23                             nameList.get(index));
24        }
25
26        // Now insert an item at index 1.
27        nameList.add(1, "Mary");
28
29        System.out.println("Mary was added at index 1. " +
30                           "Here are the items now.");
31
32        // Display the items in nameList and their indices.
33        for (int index = 0; index < nameList.size(); index++)
34        {
35          System.out.println("Index: " + index + " Name: " +
36                             nameList.get(index));
37       }
38     }
39  }
```

**Program Output**

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
Mary was added at index 1. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Mary
Index: 2 Name: Catherine
Index: 3 Name: Bill
```

Note that when a new item was added at index 1 (in line 27), the item that was previously stored at index 1 was shifted in position to index 2. When an item is added at a specific index, the items that come after it are shifted upward in position to accommodate the new item. This means that the index of each item after the new item will be increased by one.

Note that an error will occur if you call the `add` method with an invalid index.

## Replacing an Item

The `ArrayList` class's `set` method can be used to replace an item at a specific index with another item. For example, the following statement will replace the item currently at index 1 with the String "Jennifer":

```
nameList.set(1, "Jennifer");
```

This is demonstrated in the program *ArrayListDemo5.java*, which is in this chapter's source code folder, available at www.pearsonhighered.com/gaddis. Note that an error will occur if you specify an invalid index.

## Capacity

Previously you learned that an `ArrayList` object's size is the number of items stored in the `ArrayList` object. When you add an item to the `ArrayList` object, its size increases by one, and when you remove an item from the `ArrayList` object, its size decreases by one.

An `ArrayList` object also has a *capacity*, which is the number of items it can store without having to increase its size. When an `ArrayList` object is first created, using the no-arg constructor, it has an initial capacity of 10 items. This means that it can hold up to 10 items without having to increase its size. When the eleventh item is added, the `ArrayList` object must increase its size to accommodate the new item. You can specify a different starting capacity, if you desire, by passing an `int` argument to the `ArrayList` constructor. For example, the following statement creates an `ArrayList` object with an initial capacity of 100 items:

```
ArrayList<String> list = new ArrayList<String>(100);
```

## Storing Your Own Objects in an `ArrayList`

All of the examples we have looked at so far use `ArrayList` objects to hold `Strings`. You can create an `ArrayList` to hold any type of object. For example, the following statement creates an `ArrayList` that can hold `BankAccount` objects:

```
ArrayList<BankAccount> accountList = new ArrayList<BankAccount>();
```

By specifying `BankAccount` inside the angled brackets, we are declaring that the `ArrayList` can hold only `BankAccount` objects. Code Listing 7-27 demonstrates such an `ArrayList`.

**Code Listing 7-27** `(ArrayListDemo6.java)`

```
import java.util.ArrayList;

/**
   This program demonstrates how to store BankAccount
   objects in an ArrayList.
*/

public class ArrayListDemo6
{
   public static void main(String[] args)
   {
      // Create an ArrayList to hold BankAccount objects.
      ArrayList<BankAccount> list = new ArrayList<BankAccount>();

      // Add three BankAccount objects to the ArrayList.
      list.add(new BankAccount(100.0));
      list.add(new BankAccount(500.0));
      list.add(new BankAccount(1500.0));

      // Display each item.
      for (int index = 0; index < list.size(); index++)
      {
         BankAccount account = list.get(index);
         System.out.println("Account at index " + index +
                      "\nBalance: " + account.getBalance());
      }
   }
}
```

**Program Output**

```
Account at index 0
Balance: 100.0
Account at index 1
Balance: 500.0
Account at index 2
Balance: 1500.0
```

## Using the Diamond Operator for Type Inference

Beginning with Java 7, you can simplify the instantiation of an `ArrayList` by using the *diamond operator* ( <> ). For example, in this chapter you have seen several programs that create an `ArrayList` object with a statement such as this:

```
ArrayList<String> list = new ArrayList<String>();
```

Notice that the data type (in this case, `String`) appears between the angled brackets in two locations: first in the part that declares the reference variable, and then again in the part that calls the `ArrayList` constructor. Beginning with Java 7, you are no longer required to write the data type in the part of the statement that calls the `ArrayList` constructor. Instead, you can simply write a set of empty angled brackets, as shown here:

```
ArrayList<String> list = new ArrayList<>();
```

This set of empty angled brackets ( <> ) is called the diamond operator. It causes the compiler to infer the required data type from the reference variable declaration. Here is another example:

```
ArrayList<InventoryItem> list = new ArrayList<>();
```

This creates an `ArrayList` that can hold `InventoryItem` objects. Keep in mind that type inference was introduced in Java 7. If you are using an earlier version of the Java language, you will have to use the more lengthy form of the declaration statement to create an `ArrayList`.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

7.23 What `import` statement must you include in your code in order to use the `ArrayList` class?

7.24 Write a statement that creates an `ArrayList` object and assigns its address to a variable named `frogs`.

7.25 Write a statement that creates an `ArrayList` object and assigns its address to a variable named `lizards`. The `ArrayList` should be able to store `String` objects only.

7.26 How do you add items to an `ArrayList` object?

7.27 How do you remove an item from an `ArrayList` object?

7.28 How do you retrieve a specific item from an `ArrayList` object?

7.29 How do you insert an item at a specific location in an `ArrayList` object?

7.30 How do you determine an `ArrayList` object's size?

7.31 What is the difference between an `ArrayList` object's size and its capacity?

## 7.14 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Using an invalid subscript.** Java does not allow you to use a subscript value that is outside the range of valid subscripts for an array.
- **Confusing the contents of an integer array element with the element's subscript.** An element's subscript and the value stored in the element are not the same thing. The subscript identifies an element, which holds a value.
- **Causing an off-by-one error.** When processing arrays, the subscripts start at zero and end at one less than the number of elements in the array. Off-by-one errors are commonly caused when a loop uses an initial subscript of one and/or uses a maximum subscript that is equal to the number of elements in the array.
- **Using the = operator to copy an array.** Assigning one array reference variable to another with the `=` operator merely copies the address in one variable to the other. To copy an array, you should copy the individual elements of one array to another.
- **Using the == operator to compare two arrays.** You cannot use the `==` operator to compare two array reference variables and determine whether the arrays are equal. When you use the `==` operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables.
- **Reversing the row and column subscripts when processing a two-dimensional array.** When thinking of a two-dimensional array as having rows and columns, the first subscript accesses a row and the second subscript accesses a column. If you reverse these subscripts, you will access the wrong element.

## Review Questions and Exercises

### Multiple Choice and True/False

1. In an array declaration, this indicates the number of elements that the array will have.
   a. subscript
   b. size declarator
   c. element sum
   d. reference variable

2. Each element of an array is accessed by a number known as a(n) __________.
   a. subscript
   b. size declarator
   c. address
   d. specifier
3. The first subscript in an array is always __________.
   a. 1
   b. 0
   c. −1
   d. 1 less than the number of elements
4. The last subscript in an array is always __________.
   a. 100
   b. 0
   c. −1
   d. 1 less than the number of elements
5. Array bounds checking happens __________.
   a. when the program is compiled
   b. when the program is saved
   c. when the program runs
   d. when the program is loaded into memory
6. This array field holds the number of elements that the array has.
   a. `size`
   b. `elements`
   c. `length`
   d. `width`
7. This search algorithm steps through an array, comparing each item with the search value.
   a. binary search
   b. sequential search
   c. selection search
   d. iterative search
8. This search algorithm repeatedly divides the portion of an array being searched in half.
   a. binary search
   b. sequential search
   c. selection search
   d. iterative search
9. This is the typical number of comparisons performed by the sequential search on an array of N elements (assuming the search values are consistently found).
   a. 2N
   b. N
   c. $N^2$
   d. N/2

10. When initializing a two-dimensional array, you enclose each row's initialization list in __________.
    a. braces
    b. parentheses
    c. brackets
    d. quotation marks
11. To insert an item at a specific location in an `ArrayList` object, you use this method.
    a. `store`
    b. `insert`
    c. `add`
    d. `get`
12. To delete an item from an `ArrayList` object, you use this method.
    a. `remove`
    b. `delete`
    c. `erase`
    d. `get`
13. To determine the number of items stored in an `ArrayList` object, you use this method.
    a. `size`
    b. `capacity`
    c. `items`
    d. `length`
14. **True or False:** Java does not allow a statement to use a subscript that is outside the range of valid subscripts for an array.
15. **True or False:** An array's sitze declarator can be a negative integer expression.
16. **True or False:** Both of the following declarations are legal and equivalent:

    ```
    int[] numbers;
    int numbers[];
    ```
17. **True or False:** The subscript of the last element in a single-dimensional array is one less than the total number of elements in the array.
18. **True or False:** The values in an initialization list are stored in the array in the order that they appear in the list.
19. **True or False:** The Java compiler does not display an error message when it processes a statement that uses an invalid subscript.
20. **True or False:** When an array is passed to a method, the method has access to the original array.
21. **True or False:** The first size declarator in the declaration of a two-dimensional array represents the number of columns. The second size declarator represents the number of rows.
22. **True or False:** A two-dimensional array has multiple `length` fields.
23. **True or False:** An `ArrayList` automatically expands in size to accommodate the items stored in it.

## Find the Error

1.
   ```
   int[] collection = new int[-20];
   ```

2.
   ```
   int[] hours = 8, 12, 16;
   ```

3.
   ```
   int[] table = new int[10];
   for (int x = 1; x <= 10; x++)
   {
      table[x] = 99;
   }
   ```

4.
   ```
   String[] names = { "George", "Susan" };
   int totalLength = 0;
   for (int i = 0; i < names.length(); i++)
      totalLength += names[i].length;
   ```

5.
   ```
   String[] words = { "Hello", "Goodbye" };
   System.out.println(words.toUpperCase());
   ```

## Algorithm Workbench

1. The variable `names` references an integer array with 20 elements. Write a `for` loop that prints each element of the array.
2. The variables `numberArray1` and `numberArray2` reference arrays that each have 100 elements. Write code that copies the values in `numberArray1` to `numberArray2`.
3. a. Write a statement that declares a `String` array initialized with the following strings: “Einstein”, “Newton”, “Copernicus”, and “Kepler”.
   b. Write a loop that displays the contents of each element in the array that you declared in Question 3(a).
   c. Write code that displays the total length of all the strings in the array that you declared in Question 3(a).
4. In a program you need to store the populations of 12 countries.
   a. Define two arrays that may be used in parallel to store the names of the countries and their populations.
   b. Write a loop that uses these arrays to print each country’s name and its population.
5. In a program you need to store the identification numbers of ten employees (as `int` values) and their weekly gross pay (as `double` values).
   a. Define two arrays that may be used in parallel to store the 10 employee identification numbers and gross pay amounts.
   b. Write a loop that uses these arrays to print each of the employees’ identification number and weekly gross pay.
6. Declare a two-dimensional `int` array named `grades`. It should have 30 rows and 10 columns.
7. Write code that calculates the average of all the elements in the `grades` array that you declared in Question 6.

8. Look at the following array declaration:

```
int[][] numberArray = new int[9][11];
```

   a. Write a statement that assigns 145 to the first column of the first row of this array.
   b. Write a statement that assigns 18 to the last column of the last row of this array.

9. The `values` variable references a two-dimensional double array with 10 rows and 20 columns. Write code that sums all the elements in the array and stores the sum in the variable `total`.

10. An application uses a two-dimensional array declared as follows:

```
int[][] days = new int[29][5];
```

   a. Write code that sums each row in the array and displays the results.
   b. Write code that sums each column in the array and displays the results.

11. Write code that creates an `ArrayList` that can hold `String` objects. Add the names of three cars to the `ArrayList`, and then display the contents of the `ArrayList`.

## Short Answer

1. What is the difference between a size declarator and a subscript?

2. Look at the following array definition:

```
int[] values = new int[10];
```

   a. How many elements does the array have?
   b. What is the subscript of the first element in the array?
   c. What is the subscript of the last element in the array?

3. Look at the following array definition:

```
int[] values = { 4, 7, 6, 8, 2 };
```

What does each of the following code segments display?

`System.out.println(values[4]);` a. ______________________

`x = values[2] + values[3];`

`System.out.println(x);` b. ______________________

`x = ++values[1];`

`System.out.println(x);` c. ______________________

4. How do you define an array without providing a size declarator?

5. Assuming that `array1` and `array2` are both array reference variables, why is it not possible to assign the contents of the array referenced by `array2` to the array referenced by `array1` with the following statement?

```
array1 = array2;
```

6. How do you establish an array without providing a size declarator?

7. The following statement creates a `BankAccount` array:

```
BankAccount[] acc = new BankAccount[10];
```

Is it okay or not okay to execute the following statements?

```
acc[0].setBalance(5000.0);
acc[0].withdraw(100.0);
```

8. If a sequential search method is searching for a value that is stored in the last element of a 10,000-element array, how many elements will the search code have to read to locate the value?
9. Look at the following array definition:

   ```
   double[][] sales = new double[8][10];
   ```

   a. How many rows does the array have?
   b. How many columns does the array have?
   c. How many elements does the array have?
   d. Write a statement that stores a number in the last column of the last row in the array.

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Rainfall Class

Write a `RainFall` class that stores the total rainfall for each of 12 months into an array of `doubles`. The program should have methods that return the following:

- the total rainfall for the year
- the average monthly rainfall
- the month with the most rain
- the month with the least rain

Demonstrate the class in a complete program.

*Input Validation: Do not accept negative numbers for monthly rainfall figures.*

### 2. Payroll Class

Write a `Payroll` class that uses the following arrays as fields:

- **employeeId**. An array of seven integers to hold employee identification numbers. The array should be initialized with the following numbers:

  ```
  5658845 4520125 7895122 8777541
  8451277 1302850 7580489
  ```

- **hours**. An array of seven integers to hold the number of hours worked by each employee
- **payRate**. An array of seven `doubles` to hold each employee's hourly pay rate
- **wages**. An array of seven `doubles` to hold each employee's gross wages

The class should relate the data in each array through the subscripts. For example, the number in element 0 of the `hours` array should be the number of hours worked by the employee whose identification number is stored in element 0 of the `employeeId` array. That same employee's pay rate should be stored in element 0 of the `payRate` array.

In addition to the appropriate accessor and mutator methods, the class should have a method that accepts an employee's identification number as an argument and returns the gross pay for that employee.

Demonstrate the class in a complete program that displays each employee number and asks the user to enter that employee's hours and pay rate. It should then display each employee's identification number and gross wages.

*Input Validation: Do not accept negative values for hours or numbers less than 6.00 for pay rate.*

### 3. Charge Account Validation

Create a class with a method that accepts a charge account number as its argument. The method should determine whether the number is valid by comparing it to the following list of valid charge account numbers:

```
5658845     4520125     7895122     8777541     8451277     1302850
8080152     4562555     5552012     5050552     7825877     1250255
1005231     6545231     3852085     7576651     7881200     4581002
```

VideoNote
The Charge Account Validation Problem

These numbers should be stored in an array or an `ArrayList` object. Use a sequential search to locate the number passed as an argument. If the number is in the array, the method should return `true`, indicating the number is valid. If the number is not in the array, the method should return `false`, indicating the number is invalid.

Write a program that tests the class by asking the user to enter a charge account number. The program should display a message indicating whether the number is valid or invalid.

### 4. Charge Account Modification

Modify the charge account validation class that you wrote for Programming Challenge 3 so it reads the list of valid charge account numbers from a file. Use Notepad or another text editor to create the file.

### 5. Larger Than *n*

In a program, write a method that accepts two arguments: an array and a number *n*. Assume that the array contains integers. The method should display all of the numbers in the array that are greater than the number *n*.

### 6. Driver's License Exam

The local Driver's License Office has asked you to write a program that grades the written portion of the driver's license exam. The exam has 20 multiple choice questions. Here are the correct answers:

1. B
2. D
3. A
4. A
5. C
6. A
7. B
8. A
9. C
10. D
11. B
12. C
13. D
14. A
15. D
16. C
17. C
18. B
19. D
20. A

A student must correctly answer 15 of the 20 questions to pass the exam.

Write a class named `DriverExam` that holds the correct answers to the exam in an array field. The class should also have an array field that holds the student's answers. The class should have the following methods:

- `passed`. Returns true if the student passed the exam, or false if the student failed
- `totalCorrect`. Returns the total number of correctly answered questions

- `totalIncorrect`. Returns the total number of incorrectly answered questions
- `questionsMissed`. An `int` array containing the question numbers of the questions that the student missed

Demonstrate the class in a complete program that asks the user to enter a student's answers, and then displays the results returned from the `DriverExam` class's methods.

*Input Validation: Only accept the letters A, B, C, or D as answers.*

### 7. Magic 8 Ball

Write a program that simulates a Magic 8 Ball, which is a fortune-telling toy that displays a random response to a yes or no question. In the student sample programs for this book, you will find a text file named *8_ball_responses.txt*. The file contains 12 responses, such as "I don't think so," "Yes, of course!," "I'm not sure," and so forth. The program should read the responses from the file into an array or `ArrayList` object. It should prompt the user to ask a question, and then display one of the responses, randomly selected from the array or `ArrayList` object. The program should repeat until the user is ready to quit.

***Contents of 8_ball_responses.txt:***

```
Yes, of course!
Without a doubt, yes.
You can count on it.
For sure!
Ask me later.
I'm not sure.
I can't tell you right now.
I'll tell you after my nap.
No way!
I don't think so.
Without a doubt, no.
The answer is clearly NO.
```

### 8. Grade Book

A teacher has five students who have taken four tests. The teacher uses the following grading scale to assign a letter grade to a student, based on the average of his or her four test scores:

| Test Score | Letter Grade |
|---|---|
| 90–100 | A |
| 80–89 | B |
| 70–79 | C |
| 60–69 | D |
| 0–59 | F |

Write a class that uses a `String` array or an `ArrayList` object to hold the five students' names, an array of five characters to hold the five students' letter grades, and five arrays of four `doubles` each to hold each student's set of test scores. The class should have methods that return a specific student's name, the average test score, and a letter grade based on the average.

Demonstrate the class in a program that allows the user to enter each student's name and his or her four test scores. It should then display each student's average test score and letter grade.

*Input Validation: Do not accept test scores less than zero or greater than 100.*

### 9. Grade Book Modification

Modify the grade book application in Programming Challenge 8 so that it drops each student's lowest score when determining the test score averages and letter grades.

### 10. Average Steps Taken

A Personal Fitness Tracker is a wearable device that tracks your physical activity, calories burned, heart rate, sleeping patterns, etc. One common physical activity that most of these devices track is the number of steps you take each day.

If you have downloaded this book's source code from the companion Web site, you will find a file named *steps.txt* in the *Chapter 05* folder. (The companion Web site can be found at www.pearsonhighered.com/gaddis.) The *steps.txt* file contains the number of steps a person has taken each day for a year. There are 365 lines in the file, and each line contains the number of steps taken during a day. (The first line is the number of steps taken on January 1, the second line is the number of steps taken on January 2, and so forth.) Write a program that reads the file, and then displays the average number of steps taken for each month. (The data is from a year that was not a leap year, so February has 28 days.)

### 11. Array Operations

Write a program with an array that is initialized with test data. Use any primitive data type of your choice. The program should also have the following methods:

- `getTotal`. This method should accept a one-dimensional array as its argument and return the total of the values in the array.
- `getAverage`. This method should accept a one-dimensional array as its argument and return the average of the values in the array.
- `getHighest`. This method should accept a one-dimensional array as its argument and return the highest value in the array.
- `getLowest`. This method should accept a one-dimensional array as its argument and return the lowest value in the array.

Demonstrate each of the methods in the program.

### 12. 1994 Gas Prices

In the student sample programs for this book, you will find a text file named *1994_Weekly_Gas_Averages.txt*. The file contains the average gas price for each week in the year 1994. (There are 52 lines in the file. Line 1 contains the average price for week 1, line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file into an array or an `ArrayList`. The program should do the following:

- Display the lowest average price of the year, along with the week number for that price, and the name of the month in which it occurred.
- Display the highest average price of the year, along with the week number for that price, and the name of the month in which it occurred.

- Display the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.)

### 13. Sorted List of 1994 Gas Prices

*Note: This assignment can be done either as an enhancement to the program that you wrote for Programming Challenge 12, or as a stand-alone program.*

In the student sample programs for this book, you will find a text file named *1994_Weekly_Gas_Averages.txt*. The file contains the average gas price for each week in the year 1994. (There are 52 lines in the file. Line 1 contains the average price for week 1, line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file, and calculates the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.) Then, the program should create another file that lists the names of the months, along with each month's average gas price, sorted from lowest to highest.

### 14. Name Search

If you have downloaded this book's source code (the companion Web site is available at www.pearsonhighered.com/gaddis), you will find the following files in the *Chapter 07* folder:

- *GirlNames.txt* – This file contains a list of the 200 most popular names given to girls born in the United States for the years 2000 through 2009.
- *BoyNames.txt* – This file contains a list of the 200 most popular names given to boys born in the United States for the years 2000 through 2009.

Write a program that reads the contents of the two files into two separate arrays, or `ArrayLists`. The user should be able to enter a boy's name, a girl's name, or both, and the application will display messages indicating whether the names were among the most popular.

### 15. Population Data

If you have downloaded this book's source code (the companion Web site is available at www.pearsonhighered.com/gaddis), you will find a file named *USPopulation.txt* in the *Chapter 07* folder. The file contains the midyear population of the United States, in thousands, during the years 1950 through 1990. The first line in the file contains the population for 1950, the second line contains the population for 1951, and so forth.

Write a program that reads the file's contents into an array. The program should display the following data:

- The average annual change in population during the time period
- The year with the greatest increase in population during the time period
- The year with the smallest increase in population during the time period

### 16. World Series Champions

If you have downloaded this book's source code (the companion Web site is available at www.pearsonhighered.com/gaddis), you will find a file named *WorldSeriesWinners.txt*. This file contains a chronological list of the winning teams in the World Series from 1903 through 2009. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2009. Note that the World Series was not played in 1904 and 1994, so those years are skipped in the file.)

Write a program that lets the user enter the name of a team, and then displays the number of times that team has won the World Series in the time period from 1903 through 2009.

> **TIP:** Read the contents of the *WorldSeriesWinners.txt* file into an array, or an `ArrayList`. When the user enters the name of a team, the program should step through the array or `ArrayList`, counting the number of times the selected team appears.

### 17. 2D Array Operations

Write a program that creates a two-dimensional array initialized with test data. Use any primitive data type that you wish. The program should have the following methods:

- `getTotal`. This method should accept a two-dimensional array as its argument and return the total of all the values in the array.
- `getAverage`. This method should accept a two-dimensional array as its argument and return the average of all the values in the array.
- `getRowTotal`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the total of the values in the specified row.
- `getColumnTotal`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a column in the array. The method should return the total of the values in the specified column.
- `getHighestInRow`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the highest value in the specified row of the array.
- `getLowestInRow`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the lowest value in the specified row of the array.

Demonstrate each of the methods in this program.

### 18. Phone Book `ArrayList`

Write a class named `PhoneBookEntry` that has fields for a person's name and phone number. The class should have a constructor and appropriate accessor and mutator methods. Then write a program that creates at least five `PhoneBookEntry` objects and stores them in an `ArrayList`. Use a loop to display the contents of each object in the `ArrayList`.

### 19. Trivia Game

In this programming challenge, you will create a simple trivia game for two players. The program will work like this:

- Starting with player 1, each player gets a turn at answering 5 trivia questions. (There are 10 questions, 5 for each player.) When a question is displayed, four possible answers are also displayed. Only one of the answers is correct, and if the player selects the correct answer, he or she earns a point.
- After answers have been selected for all of the questions, the program displays the number of points earned by each player and declares the player with the highest number of points the winner.

You are to design a `Question` class to hold the data for a trivia question. The `Question` class should have `String` fields for the following data:

- A trivia question
- Possible answer 1
- Possible answer 2
- Possible answer 3
- Possible answer 4
- The number of the correct answer (1, 2, 3, or 4)

The `Question` class should have appropriate constructor(s), accessor, and mutator methods.

The program should create an array of 10 `Question` objects, one for each trivia question. (If you prefer, you can use an `ArrayList` instead of an array.) Make up your own trivia questions on the subject or subjects of your choice for the objects.

### 20. Lo Shu Magic Square

The Lo Shu Magic Square is a grid with 3 rows and 3 columns, shown in Figure 7-31. The Lo Shu Magic Square has the following properties:

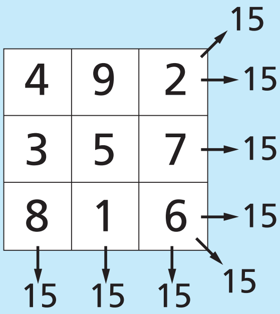
- The grid contains the numbers 1 through 9 exactly.
- The sum of each row, each column, and each diagonal all add up to the same number. This is shown in Figure 7-32.

In a program you can simulate a magic square using a two-dimensional array. Write a method that accepts a two-dimensional array as an argument, and determines whether the array is a Lo Shu Magic Square. Test the function in a program.

**Figure 7-31** Lo Shu Magic Square

| | | |
|---|---|---|
| 4 | 9 | 2 |
| 3 | 5 | 7 |
| 8 | 1 | 6 |

**Figure 7-32** Row, column, and diagonal sums in the Lo Shu Magic Square



# CHAPTER 8 A Second Look at Classes and Objects

## TOPICS

8.1 Static Class Members
8.2 Passing Objects as Arguments to Methods
8.3 Returning Objects from Methods
8.4 The `toString` Method
8.5 Writing an `equals` Method
8.6 Methods That Copy Objects
8.7 Aggregation
8.8 The `this` Reference Variable
8.9 Enumerated Types
8.10 Garbage Collection
8.11 Focus on Object-Oriented Design: Class Collaboration
8.12 Common Errors to Avoid

## 8.1 Static Class Members

**CONCEPT:** **A static class member belongs to the class, not objects instantiated from the class.**

### A Quick Review of Instance Fields and Instance Methods

Recall from Chapter 6 that each instance of a class has its own set of fields, which are known as instance fields. You can create several instances of a class and store different values in each instance's fields. For example, the `Rectangle` class that we created in Chapter 6 has a `length` field and a `width` field. Let's say that `box` references an instance of the `Rectangle` class and we execute the following statement:

```
box.setLength(10);
```

This statement stores the value 10 in the `length` field that belongs to the instance that is referenced by `box`. You can think of instance fields as belonging to a specific instance of a class.

You will also recall that classes may have instance methods as well. When you call an instance method, it performs an operation on a specific instance of the class. For example, assuming that `box` references an instance of the `Rectangle` class, look at the following statement:

```
x = box.getLength();
```

This statement calls the `getLength` method, which returns the value of the `length` field that belongs to a specific instance of the `Rectangle` class: the one referenced by `box`. Both instance fields and instance methods are associated with a specific instance of a class, and they cannot be used until an instance of the class is created.

## Static Members

It is possible to create a field or method that does not belong to any instance of a class. Such members are known as static fields and static methods. When a value is stored in a static field, it is not stored in an instance of the class. In fact, an instance of the class doesn't even have to exist in order for values to be stored in the class's static fields. Likewise, static methods do not operate on the fields that belong to any instance of the class. Instead, they can operate only on static fields. You can think of static fields and static methods as belonging to the class instead of an instance of the class. In this section, we will take a closer look at static members. First we will examine static fields.

## Static Fields

When a field is declared with the key word `static`, there will be only one copy of the field in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static field is shared by all instances of the class. For example, the `Countable` class shown in Code Listing 8-1 uses a static field to keep count of the number of instances of the class that are created.

**Code Listing 8-1** (`Countable.java`)

```
/**
   This class demonstrates a static field.
*/

public class Countable
{
   private static int instanceCount = 0;

  /**
     The constructor increments the static
     field instanceCount. This keeps track
     of the number of instances of this
     class that are created.
   */

   public Countable()
   {
```

```
18         instanceCount++;
19      }
20
21     /**
22         The getInstanceCount method returns
23         the number of instances of this class
24         that have been created.
25         @return The value in the instanceCount field.
26     */
27
28      public int getInstanceCount()
29      {
30         return instanceCount;
31      }
32   }
```

First, notice in line 7 the declaration of the static field named `instanceCount` as follows:

```
private static int instanceCount = 0;
```

A static field is created by placing the key word `static` after the access specifier and before the field's data type. Notice that we have explicitly initialized the `instanceCount` field with the value 0. This initialization takes place only once, regardless of the number of instances of the class that are created.

**NOTE:** Java automatically stores 0 in all uninitialized static member variables. The `instanceCount` field in this class is explicitly initialized so it is clear to anyone reading the code that the field starts with the value 0.

Next, look at the constructor in lines 16 through 19. The constructor uses the `++` operator to increment the `instanceCount` field. Each time an instance of the `Countable` class is created, the constructor will be called and the `instanceCount` field will be incremented. As a result, the `instanceCount` field will contain the number of instances of the `Countable` class that have been created. The `getInstanceCount` method, in lines 28 through 31, returns the value in `instanceCount`. The program in Code Listing 8-2 demonstrates this class.

**Code Listing 8-2** (`StaticDemo.java`)

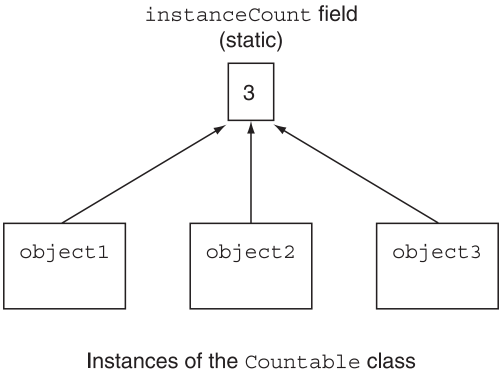
```
/**
   This program demonstrates the Countable class.
*/

public class StaticDemo
{
   public static void main(String[] args)
   {
      int objectCount;
10
11          // Create three instances of the
12          // Countable class.
13          Countable object1 = new Countable();
14          Countable object2 = new Countable();
15          Countable object3 = new Countable();
16
17          // Get the number of instances from
18          // the class's static field.
19          objectCount = object1.getInstanceCount();
20          System.out.println(objectCount +
21                             " instances of the class " +
22                             "were created.");
23       }
24    }
```

**Program Output**

```
3 instances of the class were created.
```

The program creates three instances of the `Countable` class, referenced by the variables `object1`, `object2`, and `object3`. Although there are three instances of the class, there is only one copy of the static field. This is illustrated in Figure 8-1.

**Figure 8-1** All instances of the class share the static field



In line 19 the program calls the `getInstanceCount` method to retrieve the number of instances that have been created:

```
objectCount = object1.getInstanceCount();
```

Although the program calls the `getInstanceCount` method from `object1`, the same value would be returned from any of the objects.

## Static Methods

When a class contains a static method, it isn't necessary for an instance of the class to be created in order to execute the method. The program in Code Listing 8-3 shows an example of a class with static methods.

**Code Listing 8-3** (`Metric.java`)

```java
/**
   This class demonstrates static methods.
*/

public class Metric
{
   /**
      The milesToKilometers method converts a
      distance in miles to kilometers.
      @param m The distance in miles.
      @return The distance in kilometers.
   */

   public static double milesToKilometers(double m)
   {
      return m * 1.609;
   }

   /**
      The kilometersToMiles method converts
      a distance in kilometers to miles.
      @param k The distance in kilometers.
      @return The distance in miles.
   */

   public static double kilometersToMiles(double k)
   {
      return k / 1.609;
   }
}
```

A static method is created by placing the key word `static` after the access specifier in the method header. The `Metric` class has two static methods: `milesToKilometers` and `kilometersToMiles`. Because they are declared as `static`, they belong to the class and may be called without any instances of the class being in existence. You simply write the name of the class before the dot operator in the method call. Here is an example:

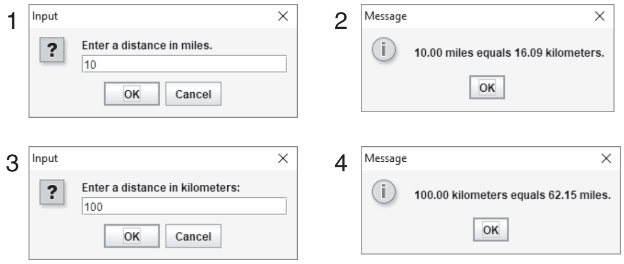
```java
kilometers = Metric.milesToKilometers(10.0);
```

This statement calls the `milesToKilometers` method, passing the value `10.0` as an argument. Notice that the method is not called from an instance of the class, but is called directly from the `Metric` class. Code Listing 8-4 shows a program that uses the `Metric` class. Figure 8-2 shows an example of interaction with the program.

**Code Listing 8-4** **(`MetricDemo.java`)**

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the Metric class.
*/

public class MetricDemo
{
   public static void main(String[] args)
   {
      String input; // To hold input
      double miles; // A distance in miles
      double kilos; // A distance in kilometers

      // Get a distance in miles.
      input = JOptionPane.showInputDialog("Enter " +
                             "a distance in miles.");
      miles = Double.parseDouble(input);

      // Convert the distance to kilometers.
      kilos = Metric.milesToKilometers(miles);
      JOptionPane.showMessageDialog(null,
        String.format("%,.2f miles equals %,.2f kilometers.",
                      miles, kilos));

      // Get a distance in kilometers.
      input = JOptionPane.showInputDialog("Enter " +
                       "a distance in kilometers: ");
      kilos = Double.parseDouble(input);

      // Convert the distance to kilometers.
      miles = Metric.kilometersToMiles(kilos);
      JOptionPane.showMessageDialog(null,
        String.format("%,.2f kilometers equals %,.2f miles.",
                      kilos, miles));

        System.exit(0);
    }
}
```

**Figure 8-2** Interaction with the `MetricDemo.java` program (Oracle Corporate Counsel)



1 Input

Enter a distance in miles.

10

2 Message

10.00 miles equals 16.09 kilometers.

3 Input

Enter a distance in kilometers:

100

4 Message

100.00 kilometers equals 62.15 miles.

Static methods are convenient for many tasks because they can be called directly from the class, as needed. They are most often used to create utility classes that perform operations on data, but have no need to collect and store data. The `Metric` class is a good example. It is used as a container to hold methods that convert miles to kilometers and vice versa, but is not intended to store any data.

The only limitation that static methods have is that they cannot refer to non-static members of the class. This means that any method called from a static method must also be static. It also means that if the method uses any of the class's fields, they must be static as well.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

8.1 What is the difference between an instance field and a static field?

8.2 What action is possible with a static method that isn't possible with an instance method?

8.3 Describe the limitation of static methods.

## 8.2 Passing Objects as Arguments to Methods

**CONCEPT:** **To pass an object as a method argument, you pass an object reference.**

In Chapter 5, we discussed how primitive values, as well as references to `String` objects, can be passed as arguments to methods. You can also pass references to other types of objects as arguments to methods. For example, recall that in Chapter 6, we developed a `Rectangle` class. The program in Code Listing 8-5 creates an instance of the `Rectangle` class and then passes a reference to that object as an argument to a method.

**Code Listing 8-5** (`PassObject.java`)

```
/**
   This program passes an object as an argument.
*/

public class PassObject
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle(12.0, 5.0);

      // Pass a reference to the object to
      // the displayRectangle method.
      displayRectangle(box);
   }

   /**
      The displayRectangle method displays the
      length and width of a rectangle.
      @param r A reference to a Rectangle
               object.
   */

   public static void displayRectangle(Rectangle r)
   {
      // Display the length and width.
      System.out.println("Length : " + r.getLength() +
                         " Width : " + r.getWidth());
   }
}
```

**Program Output**

```
Length : 12.0 Width : 5.0
```

In this program's `main` method, the `box` variable is a `Rectangle` reference variable. In line 14 its value is passed as an argument to the `displayRectangle` method. The `displayRectangle` method has a parameter variable, `r`, which is also a `Rectangle` reference variable, that receives the argument.

Recall that a reference variable holds the memory address of an object. When the `displayRectangle` method is called, the address that is stored in `box` is passed into the `r` parameter variable. This is illustrated in Figure 8-3. This means that when the `displayRectangle` method is executing, `box` and `r` both reference the same object. This is illustrated in Figure 8-4.

**Figure 8-3** Passing a reference as an argument

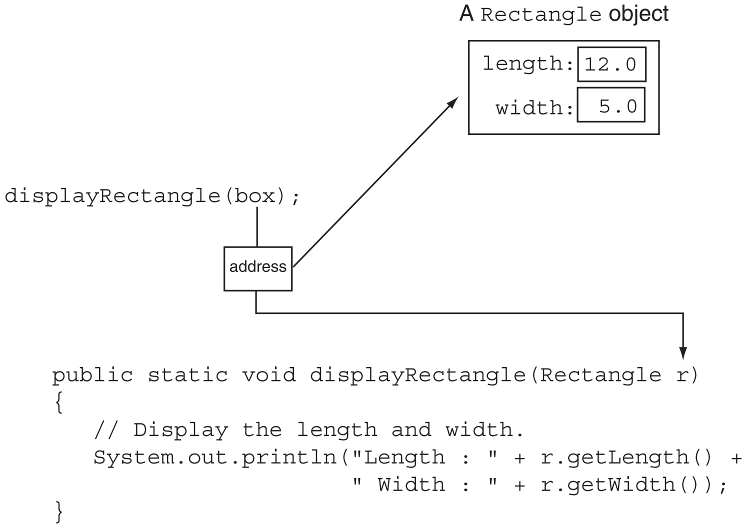
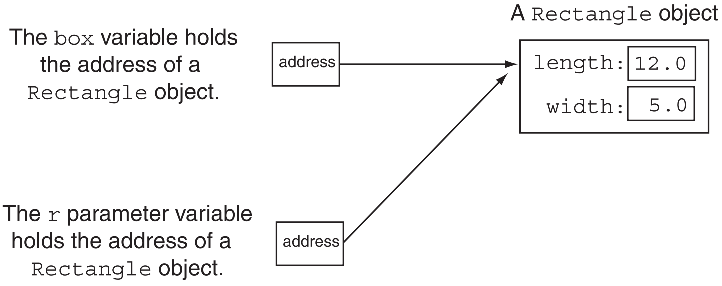

A Rectangle object
length: 12.0
width: 5.0
displayRectangle(box);
address
public static void displayRectangle(Rectangle r)
{
// Display the length and width.
System.out.println("Length : " + r.getLength() +
" Width : " + r.getWidth());
}

**Figure 8-4** Both box and r reference the same object


A Rectangle object
The box variable holds the address of a Rectangle object.
address
length: 12.0
width: 5.0
The r parameter variable holds the address of a Rectangle object.
address

Recall from Chapter 5 that when a variable is passed as an argument to a method, it is said to be passed by value. This means that a copy of the variable's value is passed into the method's parameter. When the method changes the contents of the parameter variable, it does not affect the contents of the original variable that was passed as an argument. When a reference variable is passed as an argument to a method, however, the method has access to the object that the variable references. As you can see from Figure 8-4, the `displayRectangle` method has access to the same `Rectangle` object that the `box` variable references. When a method receives an object reference as an argument, it is possible for the method to modify the contents of the object referenced by the variable. This is demonstrated in Code Listing 8-6.

**Code Listing 8-6** `(PassObject2.java)`

```
/**
   This program passes an object as an argument.
   The object is modified by the receiving method.
*/

public class PassObject2
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle(12.0, 5.0);

      // Display the object's contents.
      System.out.println("Contents of the box object:");
      System.out.println("Length : " + box.getLength() +
                         " Width : " + box.getWidth());

      // Pass a reference to the object to the
      // changeRectangle method.
      changeRectangle(box);

      // Display the object's contents again.
      System.out.println("\nNow the contents of the " +
                         "box object are:");
      System.out.println("Length : " + box.getLength() +
                         " Width : " + box.getWidth());
   }

   /**
      The changeRectangle method sets a Rectangle
      object's length and width to 0.
      @param r The Rectangle object to change.
   */

   public static void changeRectangle(Rectangle r)
   {
      r.setLength(0.0);
      r.setWidth(0.0);
   }
}
```

**Program Output**

```
Contents of the box object:
Length : 12.0 Width : 5.0

Now the contents of the box object are:
Length : 0.0 Width : 0.0
```

When writing a method that receives the value of a reference variable as an argument, you must take care not to accidentally modify the contents of the object that is referenced by the variable.

## 8.3 Returning Objects from Methods

**CONCEPT:** **A method can return a reference to an object.**

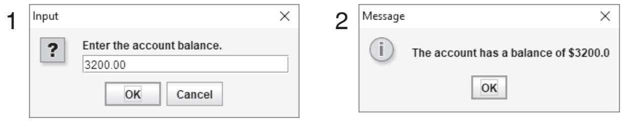
VideoNote
Returning Objects from Methods

Just as methods can be written to return an `int`, `double`, `float`, or other primitive data type, they can also be written to return a reference to an object. For example, recall the `BankAccount` class that was discussed in Chapter 6. The program in Code Listing 8-7 uses a method, `getAccount`, which returns a reference to a `BankAccount` object. Figure 8-5 shows example interaction with the program.

**Code Listing 8-7** (`ReturnObject.java`)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates how a method
   can return a reference to an object.
*/

public class ReturnObject
{
   public static void main(String[] args)
   {
      BankAccount account;

      // Get a reference to a BankAccount object.
      account = getAccount();

      // Display the account's balance.
      JOptionPane.showMessageDialog(null,
               "The account has a balance of $" +
               account.getBalance());

      System.exit(0);
   }

   /**
      The getAccount method creates a BankAccount
      object with the balance specified by the
      user.
      @return A reference to the object.
   */

   public static BankAccount getAccount()
   {
      String input;       // To hold input
      double balance;     // Account balance

      // Get the balance from the user.
      input = JOptionPane.showInputDialog("Enter " +
                              "the account balance.");
      balance = Double.parseDouble(input);

      // Create a BankAccount object and return
      // a reference to it.
      return new BankAccount(balance);
   }
}
```

**Figure 8-5** Interaction with the `ReturnObject.java` program (Oracle Corporate Counsel)

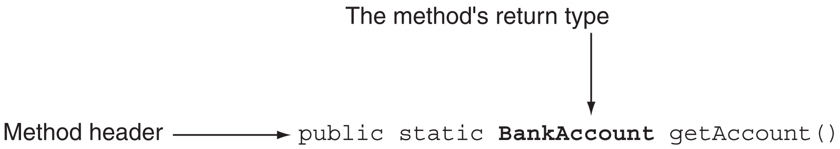

1
Input
OK
Cancel
2
Message
OK

Enter the account balance.
3200.00

The account has a balance of $3200.0

Notice that the `getAccount` method has a return data type of `BankAccount`. Figure 8-6 shows the method's return type, which is listed in the method header.

**Figure 8-6** The `getAccount` method header


The method's return type
Method header
public static BankAccount getAccount()

A return type of `BankAccount` means the method returns a reference to a `BankAccount` object when it terminates. The following statement, which appears in line 15, assigns the `getAccount` method's return value to `account`:

```
account = getAccount();
```

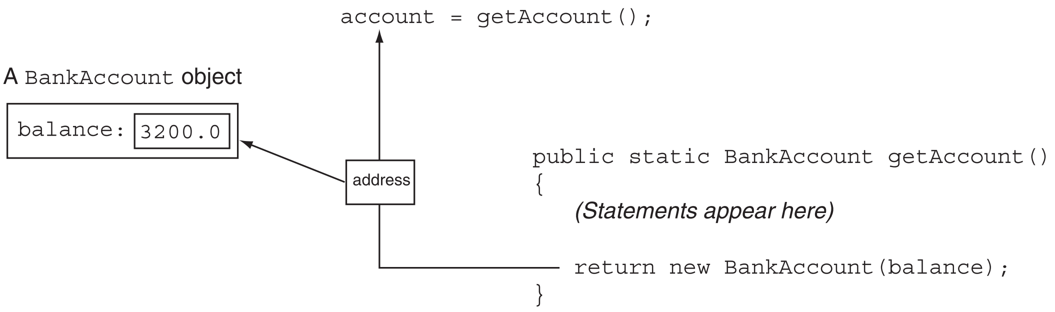
After this statement executes, the `account` variable will reference the `BankAccount` object that was returned from the `getAccount` method.

Now let's look at the `getAccount` method. In lines 38 and 39 the method uses a `JOptionPane` dialog box to get the account balance from the user. In line 40 the value entered by the user is converted to a double and assigned to `balance`, a local variable. The last statement in the method, in line 44, is the following `return` statement:

```
return new BankAccount(balance);
```

This statement uses the `new` key word to create a `BankAccount` object, passing `balance` as an argument to the constructor. The address of the object is then returned from the method, as illustrated in Figure 8-7. Back in line 15, where the method is called, the address is assigned to `account`.

**Figure 8-7** The `getAccount` method returns a reference to a `BankAccount` object



## 8.4 The `toString` Method

**CONCEPT: Most classes can benefit from having a method named `toString`, which is implicitly called under certain circumstances. Typically, the method returns a string that represents the state of an object.**

Quite often we need to display a message that indicates an object's state. An object's *state* is simply the data that is stored in the object's fields at any given moment. For example, recall that the `BankAccount` class has one field: `balance`. At any given moment, a `BankAccount` object's `balance` field will hold some value. The value of the `balance` field represents the object's state at that moment. The following might be an example of code that displays a `BankAccount` object's state:

```
BankAccount account = new BankAccount(1500.0);
System.out.println("The account balance is $" +
                   account.getBalance());
```

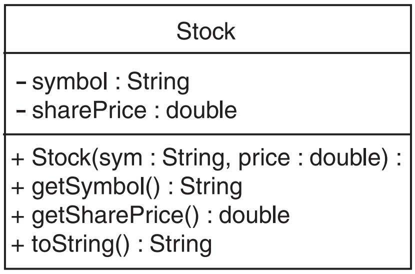
The first statement creates a `BankAccount` object, passing the value 1500.0 to the constructor. Recall that the `BankAccount` constructor stores this value in the `balance` field. After this statement executes, the `account` variable will reference the `BankAccount` object. In the second statement, the `System.out.println` method displays a string showing the value of the object's `balance` field. The output of this statement will look like this:

```
The account balance is $1500.0
```

Let's take a closer look at the second statement, which displays the state of the object. The argument that is passed to `System.out.println` is a string, which is put together from two pieces. The concatenation operator (+) joins the pieces together. The first piece is the string literal `"The account balance is $"`. To this, the value returned from the `getBalance` method is concatenated. The resulting string, which is displayed on the screen, represents the current state of the object.

Creating a string that represents the state of an object is such a common task that many programmers equip their classes with a method that returns such a string. In Java, it is standard practice to name this method `toString`. Let's look at an example of a class that has a `toString` method. Figure 8-8 shows the UML diagram for the `Stock` class, which holds data about a company's stock.

**Figure 8-8** UML diagram for the `Stock` class



This class has two fields: `symbol` and `sharePrice`. The `symbol` field holds the trading symbol for the company's stock. This is a short series of characters that are used to identify the stock on the stock exchange. For example, the XYZ Company's stock might have the trading symbol XYZ. The `sharePrice` field holds the current price per share of the stock. Table 8–1 describes the class's methods.

**Table 8-1** The `Stock` class methods

| Method | Description |
|---|---|
| Constructor | This constructor accepts arguments that are assigned to the `symbol` and `sharePrice` fields. |
| `getSymbol` | This method returns the value in the `symbol` field. |
| `getSharePrice` | This method returns the value in the `sharePrice` field. |
| `toString` | This method returns a string representing the state of the object. The string will be appropriate for displaying on the screen. |

Code Listing 8-8 shows the code for the `Stock` class. (This file is in the source code folder *Chapter 08\Stock Class Phase 1*.)

**Code Listing 8-8** (`Stock.java`)

```
/**
   The Stock class holds data about a stock.
*/

public class Stock
{
   private String symbol;        // Trading symbol of stock
   private double sharePrice;    // Current price per share

   /**
      Constructor
      @param sym The stock's trading symbol.
      @param price The stock's share price.
   */

   public Stock(String sym, double price)
   {
      symbol = sym;
      sharePrice = price;
   }

   /**
      getSymbol method
      @return The stock's trading symbol.
   */

   public String getSymbol()
   {
      return symbol;
   }

   /**
      getSharePrice method
      @return The stock's share price
   */

   public double getSharePrice()
   {
      return sharePrice;
   }

   /**
43       toString method
44       @return A string indicating the object's
45               trading symbol and share price.
46    */
47 
48    public String toString()
49    {
50       // Create a string describing the stock.
51       String str = "Trading symbol: " + symbol +
52                    "\nShare price: " + sharePrice;
53 
54       // Return the string.
55       return str;
56    }
57 }
```

The `toString` method appears in lines 48 through 56. The method creates a string listing the stock's trading symbol and price per share. This string is then returned from the method. A call to the method can then be passed to `System.out.println`, as shown in the following code:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany.toString());
```

This code would produce the following output:

```
Trading symbol: XYZ
Share price: 9.62
```

In actuality, it is unnecessary to explicitly call the `toString` method in this example. If you write a `toString` method for a class, Java will automatically call the method when the object is passed as an argument to `print` or `println`. The following code would produce the same output as that previously shown:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany);
```

Java also implicitly calls an object's `toString` method any time you concatenate an object of the class with a string. For example, the following code would implicitly call the `xyzCompany` object's `toString` method:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println("The stock data is:\n" + xyzCompany);
```

This code would produce the following output:

```
The stock data is:
Trading symbol: XYZ
Share price: 9.62
```

Code Listing 8-9 shows a complete program demonstrating the `Stock` class's `toString` method. (This file is in the source code folder *Chapter 08\Stock Class Phase 1*.)

**Code Listing 8-9** (`StockDemo1.java`)

```
/**
   This program demonstrates the Stock class's
   toString method.
*/

public class StockDemo1
{
   public static void main(String[] args)
   {
      // Create a Stock object for the XYZ Company.
      // The trading symbol is XYZ and the current
      // price per share is $9.62.
      Stock xyzCompany = new Stock ("XYZ", 9.62);

      // Display the object's values.
      System.out.println(xyzCompany);
   }
}
```

**Program Output**

```
Trading symbol: XYZ
Share price: 9.62
```

**NOTE:** Every class automatically has a `toString` method that returns a string containing the object's class name, followed by the `@` symbol, followed by an integer that is usually based on the object's memory address. This method is called when necessary if you have not provided your own `toString` method. You will learn more about this in Chapter 10.

## 8.5 Writing an `equals` Method

**CONCEPT: You cannot determine whether two objects contain the same data by comparing them with the == operator. Instead, the class must have a method such as `equals` for comparing the contents of objects.**

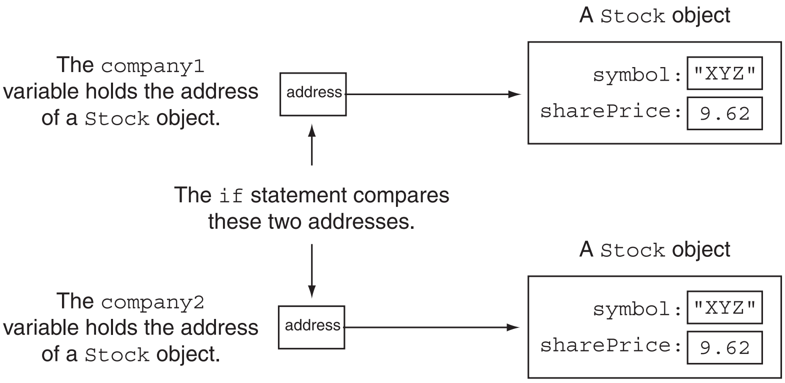
Recall from Chapter 3 that the that the `String` class has a method named `equals`, which determines whether two strings are equal. You can write an `equals` method for any of your own classes as well.

In fact, you must write an `equals` method (or one that works like it) for a class in order to determine whether two objects of the class contain the same values. This is because you cannot use the `==` operator to compare the contents of two objects. For example, the following code might appear to compare the contents of two `Stock` objects, but in reality does not:

```
// Create two Stock objects with the same values.
Stock company1 = new Stock("XYZ", 9.62);
Stock company2 = new Stock("XYZ", 9.62);

// Use the == operator to compare the objects.
// (This is a mistake.)
if (company1 == company2)
   System.out.println("Both objects are the same.");
else
   System.out.println("The objects are different.");
```

When you use the == operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables. This is illustrated in Figure 8-9.

**Figure 8-9** The `if` statement tests the contents of the reference variables, not the contents of the objects the variables reference


A Stock object
The company1 variable holds the address of a Stock object.
address
symbol: "XYZ"
sharePrice: 9.62
The if statement compares these two addresses.
A Stock object
The company2 variable holds the address of a Stock object.
address
symbol: "XYZ"
sharePrice: 9.62

Because the two variables reference different objects in memory, they will contain different addresses. Therefore, the result of the `boolean` expression `company1 == company2` is `false` and the code reports that the objects are not the same. Instead of using the == operator to compare the two `Stock` objects, we should write an `equals` method that compares the contents of the two objects.

In the source code folder *Chapter 08\Stock Class Phase 2,* you will find a revision of the `Stock` class. This version of the class has an `equals` method. The code for the method follows (no other part of the class has changed, so only the `equals` method is shown):

```
public boolean equals(Stock object2)
{
   boolean status;

   // Determine whether this object's symbol and
   // sharePrice fields are equal to object2's
```

```
      // symbol and sharePrice fields.
      if (symbol.equals(object2.symbol) &&
         sharePrice == object2.sharePrice)
         status = true;   // Yes, the objects are equal.
      else
         status = false;  // No, the objects are not equal.

      // Return the value in status.
      return status;
   }
```

The `equals` method accepts a `Stock` object as its argument. The parameter variable `object2` will reference the object that was passed as an argument. The `if` statement performs the following comparison: If the `symbol` field of the calling object is equal to the `symbol` field of `object2`, *and* the `sharePrice` field of the calling object is equal to the `sharePrice` field of `object2`, then the two objects contain the same values. In this case, the local variable `status` (a `boolean`) is set to `true`. Otherwise, `status` is set to `false`. Finally, the method returns the value of the `status` variable.

Notice that the method can access `object2`'s `symbol` and `sharePrice` fields directly. Because `object2` references a `Stock` object, and the `equals` method is a member of the `Stock` class, the method is allowed to access `object2`'s private fields.

The program in Code Listing 8-10 demonstrates the `equals` method. (This file is also stored in the source code folder *Chapter 08\Stock Class Phase 2*.)

**Code Listing 8-10** (`StockCompare.java`)

```
/**
   This program uses the Stock class's equals
   method to compare two Stock objects.
*/

public class StockCompare
{
   public static void main(String[] args)
   {
      // Create two Stock objects with the same values.
      Stock company1 = new Stock("XYZ", 9.62);
      Stock company2 = new Stock("XYZ", 9.62);

      // Use the equals method to compare the objects.
      if (company1.equals(company2))
         System.out.println("Both objects are the same.");
      else
         System.out.println("The objects are different.");
   }
}
```

**Program Output**

```
Both objects are the same.
```

If you want to be able to compare the objects of a given class, you should always write an `equals` method for the class.

**NOTE:** Every class automatically has an `equals` method, which works the same as the == operator. This method is called when necessary if you have not provided your own `equals` method. You will learn more about this in Chapter 10.

## 8.6 Methods That Copy Objects

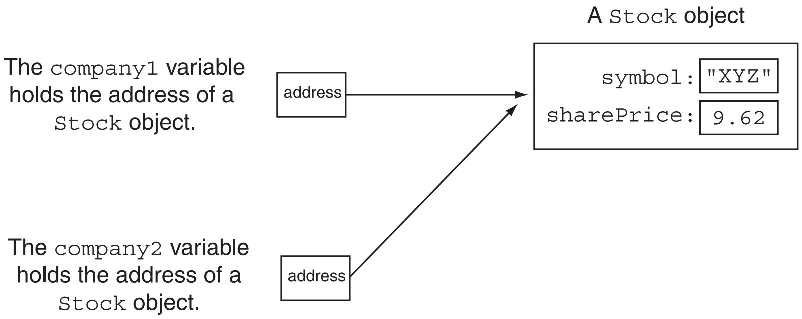
**CONCEPT: You can simplify the process of duplicating objects by equipping a class with a method that returns a copy of an object.**

You cannot make a copy of an object with a simple assignment statement, as you would with a primitive variable. For example, look at the following code:

```
Stock company1 = new Stock("XYZ", 9.62);
Stock company2 = company1;
```

The first statement creates a `Stock` object and assigns its address to the `company1` variable. The second statement assigns `company1` to `company2`. This does not make a copy of the object referenced by `company1`. Rather, it makes a copy of the address that is stored in `company1` and stores that address in `company2`. After this statement executes, both the `company1` and `company2` variables will reference the same object. This is illustrated in Figure 8-10.

**Figure 8-10** Both variables reference the same object


A Stock object
The company1 variable holds the address of a Stock object.
address
symbol: "XYZ"
sharePrice: 9.62
The company2 variable holds the address of a Stock object.
address

This type of assignment operation is called a *reference copy* because only the object's address is copied, not the actual object itself. To copy the object itself, you must create a new object and then set the new object's fields to the same values as the fields of the object that is being copied. This process can be simplified by equipping the class with a method that performs this operation. The method then returns a reference to the duplicate object.

In the source code folder *Chapter 08\Stock Class Phase 3,* you will find a revision of the `Stock` class. This version of the class has a method named `copy`, which returns a copy of a `Stock` object. The code for the method follows (no other part of the class has changed so only the `copy` method is shown):

```
public Stock copy()
{
   // Create a new Stock object and initialize it
   // with the same data held by the calling object.
   Stock copyObject = new Stock(symbol, sharePrice);

   // Return a reference to the new object.
   return copyObject;
}
```

The `copy` method creates a new `Stock` object and passes the calling object's `symbol` and `sharePrice` fields as arguments to the constructor. This makes the new object a copy of the calling object. The program in Code Listing 8-11 demonstrates the `copy` method. (This file is also stored in the source code folder *Chapter 08\Stock Class Phase 3*.)

**Code Listing 8-11** `(ObjectCopy.java)`

```
/**
   This program uses the Stock class's copy method
   to create a copy of a Stock object.
*/

public class ObjectCopy
{
   public static void main(String[] args)
   {
      // Create a Stock object.
      Stock company1 = new Stock("XYZ", 9.62);

      // Declare a Stock variable
      Stock company2;

      // Make company2 reference a copy of the object
      // referenced by company1.
      company2 = company1.copy();

      // Display the contents of both objects.
      System.out.println("Company 1:\n" + company1);
      System.out.println();
      System.out.println("Company 2:\n" + company2);

      // Confirm that we actually have two objects.
      if (company1 == company2)
```

```
         {
            System.out.println("The company1 and company2 " +
                        "variables reference the same object.");
         }
         else
         {
            System.out.println("The company1 and company2 " +
                     "variables reference different objects.");
         }
      }
   }
```

**Program Output**

```
Company 1:
Trading symbol: XYZ
Share price: 9.62

Company 2:
Trading symbol: XYZ
Share price: 9.62
The company1 and company2 variables reference different objects.
```

## Copy Constructors

Another way to create a copy of an object is to use a copy constructor. A *copy constructor* is simply a constructor that accepts an object of the same class as an argument. It makes the object that is being created a copy of the object that was passed as an argument.

In the source code folder *Chapter 08\Stock Class Phase 4*, you will find another revision of the `Stock` class. This version of the class has a copy constructor. The code for the copy constructor follows (no other part of the class has changed, so only the copy constructor is shown):

```
public Stock(Stock object2)
{
   symbol = object2.symbol;
   sharePrice = object2.sharePrice;
}
```

Notice that the constructor accepts a `Stock` object as an argument. The parameter variable `object2` will reference the object that was passed as an argument. The constructor copies the values that are in `object2`'s `symbol` and `sharePrice` fields to the `symbol` and `sharePrice` fields of the object that is being created.

The following code segment demonstrates the copy constructor. It creates a `Stock` object referenced by the variable `company1`. Then it creates another `Stock` object referenced by the variable `company2`. The object referenced by `company2` is a copy of the object referenced by `company1`.

```
// Create a Stock object.
Stock company1 = new Stock("XYZ", 9.62);
// Create another Stock object that is a copy of the company1 object.
Stock company2 = new Stock(company1);
```

## 8.7 Aggregation

**CONCEPT: Aggregation occurs when an instance of a class is a field in another class.**

In real life, objects are frequently made of other objects. A house, for example, is made of door objects, window objects, wall objects, and much more. It is the combination of all these objects that makes a house object.

VideoNote Aggregation

When designing software, it sometimes makes sense to create an object from other objects. For example, suppose you need an object to represent a course that you are taking in college. You decide to create a `Course` class, which will hold the following information:

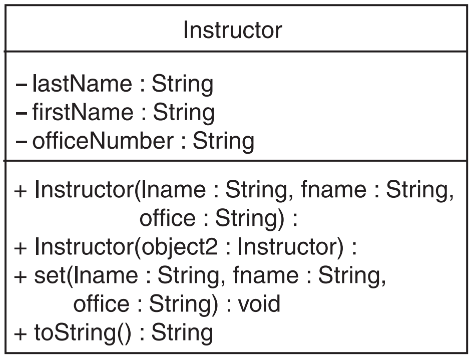
- The course name
- The instructor's last name, first name, and office number
- The textbook's title, author, and publisher

In addition to the course name, the class will hold items related to the instructor and the textbook. You could put fields for each of these items in the `Course` class. However, a good design principle is to separate related items into their own classes. In this example, an `Instructor` class could be created to hold the instructor-related data and a `TextBook` class could be created to hold the textbook-related data. Instances of these classes could then be used as fields in the `Course` class.

Let's take a closer look at how this might be done. Figure 8-11 shows a UML diagram for the `Instructor` class. To keep things simple, the class has only the following methods:

- A constructor, which accepts arguments for the instructor's last name, first name, and office number
- A copy constructor
- A `set` method, which can be used to set all of the class's fields
- A `toString` method

**Figure 8-11** UML diagram for the `Instructor` class


Instructor
– lastName : String
– firstName : String
– officeNumber : String
+ Instructor(lname : String, fname : String, office : String) :
+ Instructor(object2 : Instructor) :
+ set(lname : String, fname : String, office : String) : void
+ toString() : String

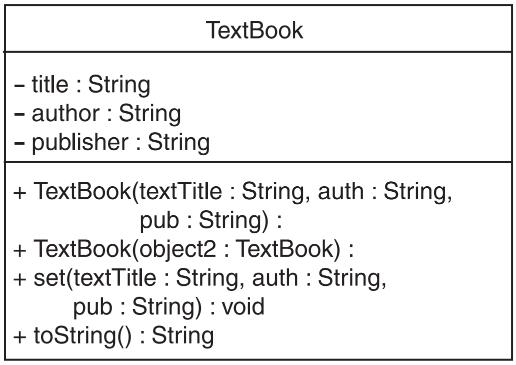
The code for the `Instructor` class is shown in Code Listing 8-12.

**Code Listing 8-12** `(Instructor.java)`

```java
/**
   This class stores data about an instructor.
*/

public class Instructor
{
   private String lastName;        // Last name
   private String firstName;       // First name
   private String officeNumber;    // Office number

   /**
      This constructor initializes the last name,
      first name, and office number.
      @param lname The instructor's last name.
      @param fname The instructor's first name.
      @param office The office number.
   */

   public Instructor(String lname, String fname,
                     String office)
   {
      lastName = lname;
      firstName = fname;
      officeNumber = office;
   }

   /**
      The copy constructor initializes the object
      as a copy of another Instructor object.
      @param object2 The object to copy.
   */

   public Instructor(Instructor object2)
   {
      lastName = object2.lastName;
      firstName = object2.firstName;
      officeNumber = object2.officeNumber;
   }

   /**
      The set method sets a value for each field.
      @param lname The instructor's last name.
      @param fname The instructor's first name.
      @param office The office number.
   */

   public void set(String lname, String fname,
                  String office)
   {
      lastName = lname;
      firstName = fname;
      officeNumber = office;
   }

   /**
      toString method
      @return A string containing the instructor
              information.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = "Last Name: " + lastName +
                   "\nFirst Name: " + firstName +
                   "\nOffice Number: " + officeNumber;

      // Return the string.
      return str;
   }
}
```

Figure 8-12 shows a UML diagram for the `TextBook` class. As before, we want to keep the class simple. The only methods it has are a constructor, a copy constructor, a `set` method, and a `toString` method. The code for the `TextBook` class is shown in Code Listing 8-13.

**Figure 8-12** UML diagram for the `TextBook` class


TextBook
- title : String
- author : String
- publisher : String
+ TextBook(textTitle : String, auth : String, pub : String) :
+ TextBook(object2 : TextBook) :
+ set(textTitle : String, auth : String, pub : String) : void
+ toString() : String

**Code Listing 8-13** `(TextBook.java)`

```
/**
   This class stores data about a textbook.
*/

public class TextBook
{
   private String title;           // Title of the book
   private String author;          // Author's last name
   private String publisher;       // Name of publisher

   /**
      This constructor initializes the title,
      author, and publisher fields
      @param textTitle The book's title.
      @param auth The author's name.
      @param pub The name of the publisher.
   */

   public TextBook(String textTitle, String auth,
                   String pub)
   {
      title = textTitle;
      author = auth;
      publisher = pub;
   }

   /**
      The copy constructor initializes the object
      as a copy of another TextBook object.
      @param object2 The object to copy.
   */

   public TextBook(TextBook object2)
   {
      title = object2.title;
      author = object2.author;
      publisher = object2.publisher;
   }

   /**
      The set method sets a value for each field.
      @param textTitle The book's title.
      @param auth The author's name.
      @param pub The name of the publisher.
   */

   public void set(String textTitle, String auth,
                   String pub)
   {
      title = textTitle;
      author = auth;
      publisher = pub;
   }

   /**
      toString method
      @return A string containing the textbook
              information.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = "Title: " + title +
                   "\nAuthor: " + author +
                   "\nPublisher: " + publisher;

      // Return the string.
      return str;
   }
}
```

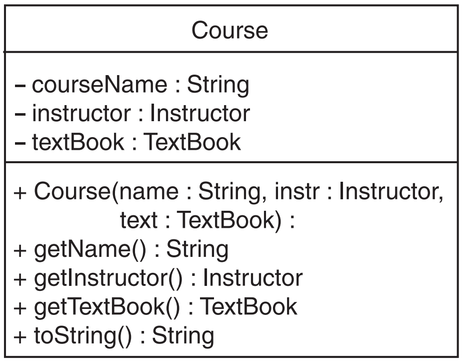
Figure 8-13 shows a UML diagram for the `Course` class. Notice that the `Course` class has an `Instructor` object and a `TextBook` object as fields. Making an instance of one class a field in another class is called *object aggregation*. The word *aggregate* means “a whole which is made of constituent parts.” In this example, the `Course` class is an aggregate class because it is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a “has a” relationship between the classes. For example, the relationships that exist among the `Course`, `Instructor`, and `TextBook` classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

The “has a” relationship is sometimes called a *whole-part relationship* because one object is part of a greater whole. The code for the `Course` class is shown in Code Listing 8-14.

**Figure 8-13** UML diagram for the `Course` class


Course
– courseName : String
– instructor : Instructor
– textBook : TextBook
+ Course(name : String, instr : Instructor,
text : TextBook) :
+ getName() : String
+ getInstructor() : Instructor
+ getTextBook() : TextBook
+ toString() : String

**Code Listing 8-14** (`Course.java`)

```
/**
   This class stores data about a course.
*/

public class Course
{
   private String courseName;     // Name of the course
   private Instructor instructor; // The instructor
   private TextBook textBook;     // The textbook

   /**
      This constructor initializes the courseName,
      instructor, and text fields.
      @param name The name of the course.
      @param instructor An Instructor object.
      @param text A TextBook object.
   */

   public Course(String name, Instructor instr,
                 TextBook text)
   {
      // Assign the courseName.
      courseName = name;

      // Create a new Instructor object, passing
      // instr as an argument to the copy constructor.
      instructor = new Instructor(instr);

      // Create a new TextBook object, passing
      // text as an argument to the copy constructor.
      textBook = new TextBook(text);
   }

   /**
      getName method
      @return The name of the course.
   */

   public String getName()
   {
      return courseName;
   }

   /**
      getInstructor method
      @return A reference to a copy of this course's
              Instructor object.
   */

   public Instructor getInstructor()
   {
      // Return a copy of the instructor object.
      return new Instructor(instructor);
   }

   /**
      getTextBook method
      @return A reference to a copy of this course's
              TextBook object.
   */

   public TextBook getTextBook()
   {
      // Return a copy of the textBook object.
      return new TextBook(textBook);
   }

   /**
      toString method
      @return A string containing the course information.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = "Course name: " + courseName +
                   "\nInstructor Information:\n" +
                   instructor +
                   "\nTextbook Information:\n" +
                   textBook;

      // Return the string.
      return str;
   }
}
```

The program in Code Listing 8-15 demonstrates the `Course` class.

**Code Listing 8-15** (`CourseDemo.java`)

```
/**
   This program demonstrates the Course class.
*/

public class CourseDemo
{
   public static void main(String[] args)
   {
      // Create an Instructor object.
      Instructor myInstructor =
          new Instructor("Kramer", "Shawn", "RH3010");

      // Create a TextBook object.
      TextBook myTextBook =
          new TextBook("Starting Out with Java",
                       "Gaddis", "Pearson");

      // Create a Course object.
      Course myCourse =
          new Course("Intro to Java", myInstructor,
                     myTextBook);

      // Display the course information.
      System.out.println(myCourse);
   }
}
```

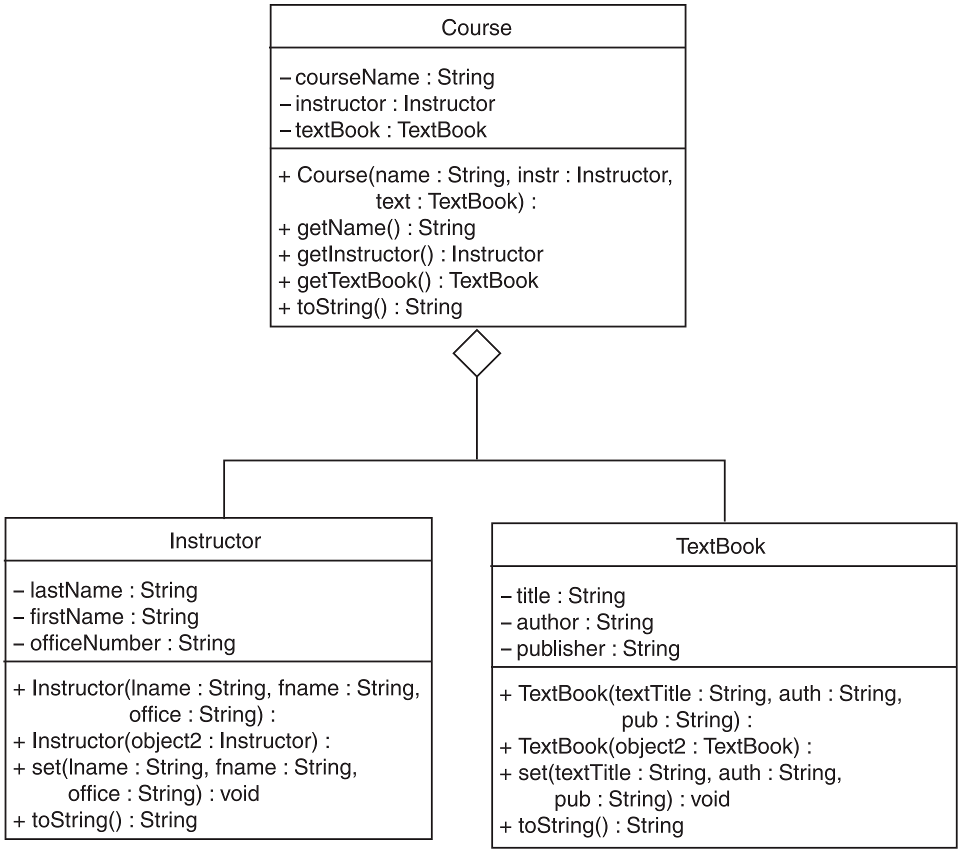
**Program Output**

```
Course name: Intro to Java
Instructor Information:
Last Name: Kramer
First Name: Shawn
Office Number: RH3010
Textbook Information:
Title: Starting Out with Java
Author: Gaddis
Publisher: Pearson
```

## Aggregation in UML Diagrams

You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that is the aggregate. Figure 8-14 is a UML diagram that shows the relationship among the `Course`, `Instructor`, and `TextBook` classes. The open diamond is closest to the `Course` class because it is the aggregate (the whole).

**Figure 8-14** UML diagram showing aggregation


Course
– courseName : String
– instructor : Instructor
– textBook : TextBook
+ Course(name : String, instr : Instructor, text : TextBook) :
+ getName() : String
+ getInstructor() : Instructor
+ getTextBook() : TextBook
+ toString() : String
Instructor
– lastName : String
– firstName : String
– officeNumber : String
+ Instructor(lname : String, fname : String, office : String) :
+ Instructor(object2 : Instructor) :
+ set(lname : String, fname : String, office : String) : void
+ toString() : String
TextBook
– title : String
– author : String
– publisher : String
+ TextBook(textTitle : String, auth : String, pub : String) :
+ TextBook(object2 : TextBook) :
+ set(textTitle : String, auth : String, pub : String) : void
+ toString() : String

## Security Issues with Aggregate Classes

When writing an aggregate class, you should be careful not to unintentionally create "security holes" that can allow code outside the class to modify private data inside the class. We will focus on the following two specific practices that can help prevent security holes in your classes:

- **Perform Deep Copies When Creating Field Objects**
  An aggregate object contains references to other objects. When you make a copy of the aggregate object, it is important that you also make copies of the objects it references. This is known as a *deep copy*. If you make a copy of an aggregate object, but only make a reference copy of the objects it references, then you have performed a *shallow copy*.

- **Return Copies of Field Objects, Not the Originals**
  When a method in the aggregate class returns a reference to a field object, return a reference to a copy of the field object.

Let's discuss each of these practices in more depth.

### Perform Deep Copies When Creating Field Objects

Let's take a closer look at the `Course` class. First, notice the arguments that the constructor accepts in lines 19 and 20 as follows:

- A reference to a `String` containing the name of the course is passed into the `name` parameter.
- A reference to an `Instructor` object is passed into the `instr` parameter.
- A reference to a `TextBook` object is passed into the `text` parameter.

Next, notice that the constructor does not merely assign `instr` to the `instructor` field. Instead, in line 27 it creates a new `Instructor` object for the `instructor` field and passes `instr` to the copy constructor. Here is the statement:

```
instructor = new Instructor(instr);
```

This statement creates a copy of the object referenced by `instr`. The `instructor` field will reference the copy.

When a class has a field that is an object, it is possible that a shallow copy operation will create a security hole. For example, suppose the `Course` constructor had been written as follows:

```
// Bad constructor!
public Course(String name, Instructor instr, TextBook text)
{
   // Assign the courseName.
   courseName = name;

   // Assign the instructor (shallow copy)
   instructor = instr; // Causes security hole!

   // Assign the textBook (shallow copy)
   textBook = text;   // Causes security hole!
}
```

In this example, the `instructor` and `textBook` fields are merely assigned the addresses of the objects passed into the constructor. This can cause problems because there may be variables outside the `Course` object that also contain references to these `Instructor` and `TextBook` objects. These outside variables would provide direct access to the `Course` object's private data.

At this point you might be wondering why a deep copy was not also done for the `courseName` field. In line 23 the `Course` constructor performs a shallow copy, simply assigning the address of the `String` object referenced by `name` to the `courseName` field. This is permissible because `String` objects are immutable. An *immutable* object does not provide a way to change its contents. Even if variables outside the `Course` class reference the same object that `courseName` references, the object cannot be changed.

### Return Copies of Field Objects, Not the Originals

When a method in an aggregate class returns a reference to a field object, it should return a reference to a copy of the field object, not the field object itself. For example, look at the `getInstructor` method in the `Course` class. The code is shown here:

```
public Instructor getInstructor()
{
   // Return a copy of the instructor object.
   return new Instructor(instructor);
}
```

Notice that the `return` statement uses the `new` key word to create a new `Instructor` object, passing the `instructor` field to the copy constructor. The object that is created is a copy of the object referenced by `instructor`. The address of the copy is then returned. This is preferable to simply returning a reference to the field object itself. For example, suppose the method had been written this way:

```
// Bad method
public Instructor getInstructor()
{
   // Return a reference to the instructor object.
   return instructor;   // WRONG! Causes a security hole.
}
```

This method returns the value stored in the `instructor` field, which is the address of an `Instructor` object. Any variable that receives the address can then access the `Instructor` object. This means that code outside the `Course` object can change the values held by the `Instructor` object. This is a security hole because the `Instructor` object is a private field! Only code inside the `Course` class should be allowed to access it.

**NOTE:** It is permissible to return a reference to a `String` object, even if the `String` object is a private field. This is because `String` objects are immutable.

## Avoid Using `null` References

By default, a reference variable that is an instance field is initialized to the value `null`. This indicates that the variable does not reference an object. Because a `null` reference variable does not reference an object, you cannot use it to perform an operation that would require the existence of an object. For example, a `null` reference variable cannot be used to call a method. If you attempt to perform an operation with a `null` reference variable, the program will terminate. For example, look at the `FullName` class shown in Code Listing 8-16.

**Code Listing 8-16** (`FullName.java`)

```java
/**
   This class stores a person's first, last, and middle
   names. The class is dangerous because it does not
   prevent operations on null reference fields.
*/

public class FullName
{
   private String lastName;        // Last name
   private String firstName;       // First name
   private String middleName;      // Middle name

   /**
      The setLastName method sets the lastName field.
      @param str The String to set lastName to.
   */

   public void setLastName(String str)
   {
      lastName = str;
   }

   /**
      The setFirstName method sets the firstName field.
      @param str The String to set firstName to.
   */

   public void setFirstName(String str)
   {
      firstName = str;
   }

   /**
      The setMiddleName method sets the middleName field.
      @param str The String to set middleName to.
   */

   public void setMiddleName(String str)
   {
      middleName = str;
   }

   /**
      The getLength method returns the length of the
      full name.
46       @return The length.
47    */
48 
49    public int getLength()
50    {
51       return lastName.length() + firstName.length()
52              + middleName.length();
53    }
54 
55    /**
56       The toString method returns the full name.
57       @return A reference to a String.
58    */
59 
60    public String toString()
61    {
62       return firstName + " " + middleName + " "
63              + lastName;
64    }
65 }
```

First, notice that the class has three `String` reference variables as fields: `lastName`, `firstName`, and `middleName`. Second, notice that the class does not have a programmer-defined constructor. When an instance of this class is created, the `lastName`, `firstName`, and `middleName` fields will be initialized to `null` by the default constructor. Third, notice that the `getLength` method uses the `lastName`, `firstName`, and `middleName` variables to call the `String` class's `length` method in lines 51 and 52. Nothing is preventing the `length` method from being called while any or all of these reference variables are set to `null`. The program in Code Listing 8-17 demonstrates this.

**Code Listing 8-17** **(`NameTester.java`)**

```
/**
   This program creates a FullName object, and then
   calls the object's getLength method before values
   are established for its reference fields. As a
   result, this program will crash.
*/

public class NameTester
{
   public static void main(String[] args)
   {
      int len; // To hold the name length

      // Create a FullName object.
      FullName name = new FullName();

      // Get the length of the full name.
      len = name.getLength();
   }
}
```

This program will crash when you run it because the `getLength` method is called before the `name` object's fields are made to reference `String` objects. One way to prevent the program from crashing is to use `if` statements in the `getLength` method to determine whether any of the fields are set to `null`. Here is an example:

```
public int getLength()
{
   int len = 0;

   if (lastName != null)
      len += lastName.length();

   if (firstName != null)
      len += firstName.length();

   if (middleName != null)
      len += middleName.length();

   return len;
}
```

Another way to handle this problem is to write a no-arg constructor that assigns values to the reference fields. Here is an example:

```
public FullName()
{
   lastName = "";
   firstName = "";
   middleName = "";
}
```

## 8.8 The `this` Reference Variable

**CONCEPT: The `this` key word is the name of a reference variable that an object can use to refer to itself. It is available to all non-static methods.**

The key word `this` is the name of a reference variable that an object can use to refer to itself. For example, recall the `Stock` class presented earlier in this chapter. The class has the following `equals` method that compares the calling `Stock` object to another `Stock` object that is passed as an argument:

```
public boolean equals(Stock object2)
{
   boolean status;

   // Determine whether this object's symbol and
   // sharePrice fields are equal to object2's
   // symbol and sharePrice fields.
   if (symbol.equals(object2.symbol) &&
       sharePrice == object2.sharePrice)
      status = true; // Yes, the objects are equal.
   else
      status = false; // No, the objects are not equal.

   // Return the value in status.
   return status;
}
```

When this method is executing, the `this` variable contains the address of the calling object. We could rewrite the `if` statement as follows, and it would perform the same operation (the changes appear in bold):

```
if (this.symbol.equals(object2.symbol) &&
    this.sharePrice == object2.sharePrice)
```

The `this` reference variable is available to all of a class's non-static methods.

## Using `this` to Overcome Shadowing

One common use of the `this` key word is to overcome the shadowing of a field name by a parameter name. Recall from Chapter 6 that if a method's parameter has the same name as a field in the same class, then the parameter name shadows the field name. For example, look at the constructor in the `Stock` class:

```
public Stock(String sym, double price)
{
   symbol = sym;
   sharePrice = price;
}
```

This method uses the parameter `sym` to accept an argument that is assigned to the `symbol` field, and the parameter `price` to accept an argument that is assigned to the `sharePrice` field. Sometimes it is difficult (and even time-consuming) to think of a good parameter name that is different from a field name. To avoid this problem, many programmers give parameters the same names as the fields to which they correspond, and then use the `this` key word to refer to the field names. For example, the `Stock` class's constructor could be written as follows:

```
public Stock(String symbol, double sharePrice)
{
   this.symbol = symbol;
   this.sharePrice = sharePrice;
}
```

Although the parameter names `symbol` and `sharePrice` shadow the field names `symbol` and `sharePrice`, the `this` key word overcomes the shadowing. Because `this` is a reference to the calling object, the expression `this.symbol` refers to the calling object's `symbol` field, and the expression `this.sharePrice` refers to the calling object's `sharePrice` field.

## Using `this` to Call an Overloaded Constructor from Another Constructor

You already know that a constructor is automatically called when an object is created. You also know that you cannot call a constructor explicitly, as you do other methods. However, there is one exception to this rule: You can use the `this` key word to call one constructor from another constructor in the same class.

To illustrate this, recall the `Stock` class that was presented earlier in this chapter. It has the following constructor:

```
public Stock(String sym, double price)
{
   symbol = sym;
   sharePrice = price;
}
```

This constructor accepts arguments that are assigned to the `symbol` and `sharePrice` fields. Let's suppose we also want a constructor that only accepts an argument for the `symbol` field, and assigns 0.0 to the `sharePrice` field. Here's one way to write the constructor:

```
public Stock(String sym)
{
   this(sym, 0.0);
}
```

This constructor simply uses the `this` variable to call the first constructor. It passes the value in `sym` as the first argument, and 0.0 as the second argument. The result is that the `symbol` field is assigned the value in `sym` and the `sharePrice` field is assigned 0.0.

Remember the following rules about using `this` to call a constructor:

- `this` can only be used to call a constructor from another constructor in the same class.
- It *must* be the first statement in the constructor that is making the call. If it is not the first statement, a compiler error will result.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

8.4 Look at the following code. (You might want to review the `Stock` class presented earlier in this chapter.)

```
Stock stock1 = new Stock("XYZ", 9.65);
Stock stock2 = new Stock("SUNW", 7.92);
```

While the `equals` method is executing as a result of the following statement, what object does `this` reference?

```
if (stock2.equals(stock1))
   System.out.println("The stocks are the same.");
```

## 8.9 Enumerated Types

**CONCEPT: An enumerated data type consists of a set of predefined values. You can use the data type to create variables that can hold only the values that belong to the enumerated data type.**

You've already learned the concept of data types and how they are used with primitive variables. For example, a variable of the `int` data type can hold integer values within a certain range. You cannot assign floating-point values to an `int` variable because only `int` values may be assigned to `int` variables. A data type defines the values that are legal for any variable of that data type.

Sometimes it is helpful to create your own data type that has a specific set of legal values. For example, suppose you wanted to create a data type named `Day`, and the legal values in that data type were the names of the days of the week (Sunday, Monday, and so forth). When you create a variable of the `Day` data type, you can only store the names of the days of the week in that variable. Any other values would be illegal. In Java, such a type is known as an *enumerated data type*.

You use the `enum` key word to create your own data type and specify the values that belong to that type. Here is an example of an enumerated data type declaration:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
           THURSDAY, FRIDAY, SATURDAY }
```

An enumerated data type declaration begins with the key word `enum`, followed by the name of the type, followed by a list of identifiers inside braces. The example declaration creates an enumerated data type named `Day`. The identifiers `SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, and `SATURDAY`, which are listed inside the braces, are known as ***enum** constants*. They represent the values that belong to the `Day` data type. Here is the general format of an enumerated type declaration:

```
enum TypeName  {  One or more enum constants  }
```

Note that the `enum` constants are not enclosed in quotation marks; therefore, they are not strings. `enum` constants must be legal Java identifiers.

**TIP:** When making up names for `enum` constants, it is not required that they be written in all uppercase letters. We could have written the `Day` type's `enum` constants as `sunday`, `monday`, and so forth. Because they represent constant values, however, the standard convention is to write them in all uppercase letters.

Once you have created an enumerated data type in your program, you can declare variables of that type. For example, the following statement declares `workDay` as a variable of the `Day` type:

```
Day workDay;
```

Because `workDay` is a `Day` variable, the only values that we can legally assign to it are the `enum` constants `Day.SUNDAY`, `Day.MONDAY`, `Day.TUESDAY`, `Day.WEDNESDAY`, `Day.THURSDAY`, `Day.FRIDAY`, and `Day.SATURDAY`. If we try to assign any value other than one of the `Day` type's `enum` constants, a compiler error will result. For example, the following statement assigns the value `Day.WEDNESDAY` to the `workDay` variable:

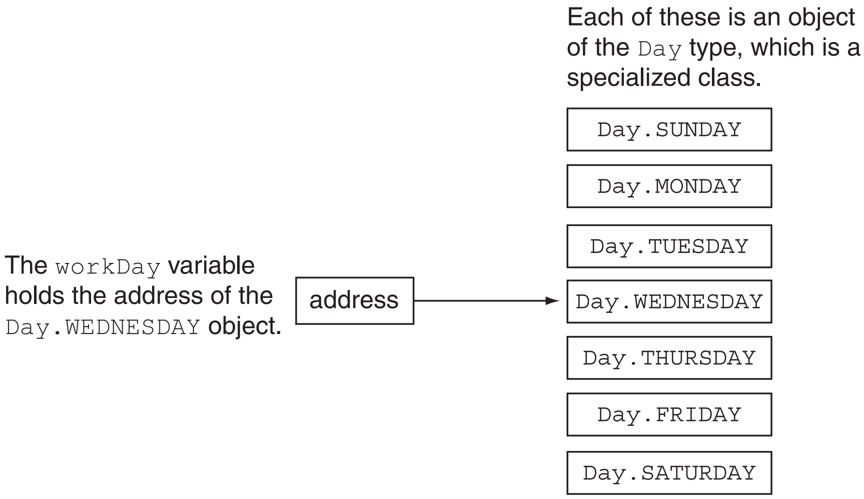
```
Day workDay = Day.WEDNESDAY;
```

Notice that we assigned `Day.WEDNESDAY` instead of just `WEDNESDAY`. The name `Day.WEDNESDAY` is the *fully qualified name* of the `Day` type's `WEDNESDAY` constant. Under most circumstances you must use the fully qualified name of an `enum` constant.

## Enumerated Types Are Specialized Classes

When you write an enumerated type declaration, you are actually creating a special kind of class. In addition, the `enum` constants that you list inside the braces are actually objects of the class. In the previous example, `Day` is a class, and the `enum` constants `Day.SUNDAY`, `Day.MONDAY`, `Day.TUESDAY`, `Day.WEDNESDAY`, `Day.THURSDAY`, `Day.FRIDAY`, and `Day.SATURDAY` are all instances of the `Day` class. When we assigned `Day.WEDNESDAY` to the `workDay` variable, we were assigning the address of the `Day.WEDNESDAY` object to the variable. This is illustrated in Figure 8-15.

**Figure 8-15** The `workDay` variable references the `Day.WEDNESDAY` object

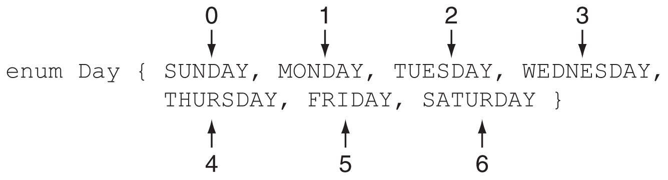

Each of these is an object of the Day type, which is a specialized class.
Day.SUNDAY
Day.MONDAY
Day.TUESDAY
The workDay variable holds the address of the Day.WEDNESDAY object.
address
Day.WEDNESDAY
Day.THURSDAY
Day.FRIDAY
Day.SATURDAY

`enum` constants, which are actually objects, come automatically equipped with a few methods. One of them is the `toString` method. The `toString` method simply returns the name of the calling `enum` constant as a string. For example, assuming that the `Day` type has been declared as previously shown, both of the following code segments display the string `WEDNESDAY` (recall that the `toString` method is implicitly called when an object is passed to `System.out.println`):

```
// This code displays WEDNESDAY.
Day workDay = Day.WEDNESDAY;
System.out.println(workDay);

// This code also displays WEDNESDAY.
System.out.println(Day.WEDNESDAY);
```

`enum` constants also have a method named `ordinal`. The `ordinal` method returns an integer value representing the constant's ordinal value. The constant's *ordinal value* is its position in the `enum` declaration, with the first constant being at position 0. Figure 8-16 shows the ordinal value of each of the constants declared in the `Day` data type.

**Figure 8-16** The Day enumerated data type and the ordinal positions of its enum constants


0
1
2
3
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY }
4
5
6

For example, assuming that the `Day` type has been declared as previously shown, look at the following code segment:

```
Day lastWorkDay = Day.FRIDAY;
System.out.println(lastWorkDay.ordinal());
System.out.println(Day.MONDAY.ordinal());
```

The ordinal value for `Day.FRIDAY` is 5 and the ordinal value for `Day.MONDAY` is 1, so this code will display:

```
5
1
```

The last enumerated data type methods that we will discuss here are `equals` and `compareTo`. The `equals` method accepts an object as its argument and returns `true` if that object is equal to the calling `enum` constant. For example, assuming that the `Day` type has been declared as previously shown, the following code segment will display "The two are the same":

```
Day myDay = Day.TUESDAY;
if (myDay.equals(Day.TUESDAY))
   System.out.println("The two are the same.");
```

The `compareTo` method is designed to compare `enum` constants of the same type. It accepts an object as its argument and returns the following:

- a negative integer value if the calling `enum` constant's ordinal value is less than the argument's ordinal value
- zero if the calling `enum` constant is the same as the argument
- a positive integer value if the calling `enum` constant's ordinal value is greater than the argument's ordinal value

For example, assuming that the `Day` type has been declared as previously shown, the following code segment will display `"FRIDAY is greater than MONDAY"`:

```
Day myDay = Day.FRIDAY;
if (myDay.compareTo(Day.MONDAY) > 0)
   System.out.println(myDay + " is greater than "
                            + Day.MONDAY);
```

One place to declare an enumerated type is inside a class. If you declare an enumerated type inside a class, it cannot be inside a method. Code Listing 8-18 shows an example. It demonstrates the `Day` enumerated type.

**Code Listing 8-18** (`EnumDemo.java`)

```
/**
   This program demonstrates an enumerated type.
*/

public class EnumDemo
{
   // Declare the Day enumerated type.
   enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
              THURSDAY, FRIDAY, SATURDAY }

   public static void main(String[] args)
   {
      // Declare a Day variable and assign it a value.
      Day workDay = Day.WEDNESDAY;

      // The following statement displays WEDNESDAY.
      System.out.println(workDay);

      // The following statement displays the ordinal
      // value for Day.SUNDAY, which is 0.
      System.out.println("The ordinal value for " +
                         Day.SUNDAY + " is " +
                         Day.SUNDAY.ordinal());

      // The following statement displays the ordinal
      // value for Day.SATURDAY, which is 6.
      System.out.println("The ordinal value for " +
                         Day.SATURDAY + " is " +
                         Day.SATURDAY.ordinal());

      // The following statement compares two enum constants.
      if (Day.FRIDAY.compareTo(Day.MONDAY) > 0)
         System.out.println(Day.FRIDAY + " is greater than " +
                            Day.MONDAY);
      else
         System.out.println(Day.FRIDAY + " is NOT greater than " +
                            Day.MONDAY);
   }
}
```

**Program Output**

```
WEDNESDAY
The ordinal value for SUNDAY is 0
The ordinal value for SATURDAY is 6
FRIDAY is greater than MONDAY
```

You can also write an enumerated type declaration inside its own file. If you do, the filename must match the name of the type. For example, if we stored the `Day` type in its own file, we would name the file *Day.java*. This makes sense because enumerated data types are specialized classes. For example, look at Code Listing 8-19. This file, *CarType.java*, contains the declaration of an enumerated data type named `CarType`. When it is compiled, a byte code file named *CarType.class* will be generated.

**Code Listing 8-19** (`CarType.java`)

```
/**
   CarType enumerated data type
*/

enum CarType { PORSCHE, FERRARI, JAGUAR }
```

Also look at Code Listing 8-20. This file, *CarColor.java*, contains the declaration of an enumerated data type named `CarColor`. When it is compiled, a byte code file named *CarColor.class* will be generated.

**Code Listing 8-20** (`CarColor.java`)

```
/**
   CarColor enumerated data type
*/

enum CarColor { RED, BLACK, BLUE, SILVER }
```

Code Listing 8-21 shows the `SportsCar` class, which uses these enumerated types. Code Listing 8-22 demonstrates the class.

**Code Listing 8-21** `(SportsCar.java)`

```
/**
   SportsCar class
*/

public class SportsCar
{
   private CarType make;    // The car's make
   private CarColor color;  // The car's color
   private double price;    // The car's price

   /**
      The constructor initializes the car's make,
      color, and price.
      @param aMake The car's make.
      @param aColor The car's color.
      @param aPrice The car's price.
   */

   public SportsCar(CarType aMake, CarColor aColor, double aPrice)
   {
      make = aMake;
      color = aColor;
      price = aPrice;
   }

   /**
      getMake method
      @return The car's make.
   */

   public CarType getMake()
   {
      return make;
   }

   /**
      getColor method
      @return The car's color.
   */

   public CarColor getColor()
   {
      return color;
   }

   /**
      getPrice method
      @return The car's price.
   */

   public double getPrice()
   {
      return price;
   }

   /**
      toString method
      @return A string indicating the car's make,
              color, and price.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = String.format("Make: %s\nColor: %s\nPrice: $%,.2f",
                                 make, color, price);

      // Return the string.
      return str;
   }
}
```

**Code Listing 8-22** (SportsCarDemo.java)

```
/**
   This program demonstrates the SportsCar class.
*/

public class SportsCarDemo
{
   public static void main(String[] args)
   {
      // Create a SportsCar object.
      SportsCar yourNewCar = new SportsCar(CarType.PORSCHE,
                                           CarColor.RED, 100000);

      // Display the object's values.
      System.out.println(yourNewCar);
   }
}
```

**Program Output**

```
Make: PORSCHE
Color: RED
Price: $100,000.00
```

## Switching On an Enumerated Type

Java allows you to test an `enum` constant with a `switch` statement. For example, look at the program in Code Listing 8-23. It creates a `SportsCar` object, and then uses a `switch` statement to test the object's `make` field.

**Code Listing 8-23** `(SportsCarDemo2.java)`

```
/**
   This program shows that you can switch on an
   enumerated type.
*/

public class SportsCarDemo2
{
   public static void main(String[] args)
   {
      // Create a SportsCar object.
      SportsCar yourNewCar = new SportsCar(CarType.PORSCHE,
                                        CarColor.RED, 100000);

      // Get the car make and switch on it.
      switch (yourNewCar.getMake())
      {
         case PORSCHE :
            System.out.println("Your car was made in Germany.");
            break;
         case FERRARI :
```

```
21             System.out.println("Your car was made in Italy.");
22             break;
23          case JAGUAR :
24             System.out.println("Your car was made in England.");
25             break;
26          default:
27             System.out.println("I'm not sure where that car "
28                                + "was made.");
29       }
30    }
31 }
```

**Program Output**

```
Your car was made in Germany.
```

In line 15 the `switch` statement tests the value returned from the `yourNewCar.getMake()` method. This method returns a `CarType` enumerated constant. Based upon the value returned from the method, the program then branches to the appropriate `case` statement. Notice in the `case` statements that the enumerated constants are not fully qualified. In other words, we had to write `PORSCHE`, `FERRARI`, and `JAGUAR` instead of `CarType.PORSCHE`, `CarType.FERRARI`, and `CarType.JAGUAR`. If you give a fully qualified `enum` constant name as a `case` expression, a compiler error will result.

**TIP:** Notice that the `switch` statement in Code Listing 8-23 has a `default` section, even though it has a `case` statement for every `enum` constant in the `CarType` type. This will handle things in the event that more `enum` constants are added to the *CarType* file. This type of planning is an example of "defensive programming."

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

8.5 Look at the following statement, which declares an enumerated data type:

```
enum Flower { ROSE, DAISY, PETUNIA }
```

a) What is the name of the data type?

b) What is the ordinal value for the `enum` constant `ROSE`? For `DAISY`? For `PETUNIA`?

c) What is the fully qualifed name of the `enum` constant `ROSE`? Of `DAISY`? Of `PETUNIA`?

d) Write a statement that declares a variable of this enumerated data type. The variable should be named `flora`. Initialize the variable with the `PETUNIA` constant.

8.6 Assume that the following enumerated data type has been declared:

```
enum Creatures{ HOBBIT, ELF, DRAGON }
```

What will the following code display?

```
System.out.println(Creatures.HOBBIT + " "
                   + Creatures.ELF + " "
                   + Creatures.DRAGON);
```

8.7 Assume that the following enumerated data type has been declared:

```
enum Letters { Z, Y, X }
```

What will the following code display?

```
if (Letters.Z.compareTo(Letters.X) > 0)
   System.out.println("Z is greater than X.");
else
   System.out.println("Z is not greater than X.");
```

## 8.10 Garbage Collection

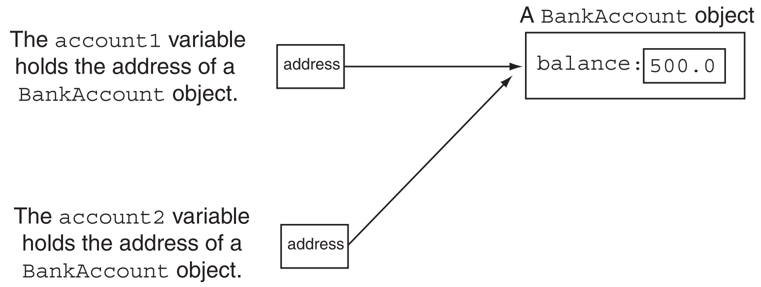
**CONCEPT: The Java Virtual Machine periodically runs a process known as the garbage collector, which removes unreferenced objects from memory.**

When an object is no longer needed, it should be destroyed so the memory it uses can be freed for other purposes. Fortunately, you do not have to destroy objects after you are finished using them. The Java Virtual Machine periodically performs a process known as garbage collection, which automatically removes unreferenced objects from memory. For example, look at the following code:

```
// Declare two BankAccount reference variables.
BankAccount account1, account2;

// Create an object and reference it with account1.
account1 = new BankAccount(500.0);

// Reference the same object with account2.
account2 = account1;

// Store null in account1 so it no longer
// references the object.
account1 = null;

// The object is still referenced by account2, though.
// Store null in account2 so it no longer references
// the object.
account2 = null;

// Now the object is no longer referenced, so it
// can be removed by the garbage collector.
```

This code uses two reference variables, `account1` and `account2`. A `BankAccount` object is created and referenced by `account1`. Then, `account1` is assigned to `account2`, which causes `account2` to reference the same object as `account1`. This is illustrated in Figure 8-17.

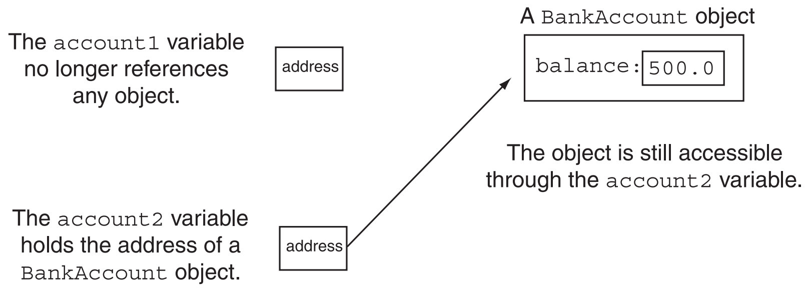
**Figure 8-17** Both `account1` and `account2` reference the same object



A BankAccount object
balance: 500.0
The account1 variable holds the address of a BankAccount object.
address
The account2 variable holds the address of a BankAccount object.
address

Next, the `null` value is assigned to `account1`. This removes the address of the object from the `account1` variable, causing it to no longer reference the object. Figure 8-18 illustrates this.

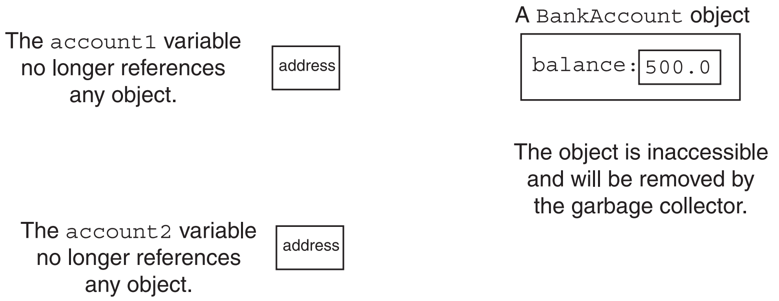
**Figure 8-18** The object is only referenced by the `account2` variable



A BankAccount object
balance: 500.0
The account1 variable no longer references any object.
address
The object is still accessible through the account2 variable.
The account2 variable holds the address of a BankAccount object.
address

The object is still accessible, however, because it is referenced by the `account2` variable. The next statement assigns `null` to `account2`. This removes the object's address from `account2`, causing it to no longer reference the object. Figure 8-19 illustrates this. Because the object is no longer accessible, it will be removed from memory the next time the garbage collector process runs.

**Figure 8-19** The object is no longer referenced



A BankAccount object
balance: 500.0
The account1 variable no longer references any object.
address
The object is inaccessible and will be removed by the garbage collector.
The account2 variable no longer references any object.
address

## The `finalize` Method

If a class has a method named `finalize`, it is called automatically just before an instance of the class is destroyed by the garbage collector. If you wish to execute code just before an object is destroyed, you can create a `finalize` method in the class and place the code there. The `finalize` method accepts no arguments and has a `void` return type.

> **NOTE:** The garbage collector runs periodically, and you cannot predict exactly when it will execute. Therefore, you cannot know exactly when an object's finalize method will execute.

## 8.11 Focus on Object-Oriented Design: Class Collaboration

**CONCEPT: It is common for classes to interact, or collaborate, with each other to perform their operations. Part of the object-oriented design process is identifying the collaborations among classes.**

In an object-oriented application it is common for objects of different classes to collaborate. This simply means that objects interact with each other. Sometimes one object will need the services of another object in order to fulfill its responsibilities. For example, let's say an object needs to read values from the keyboard, and then write those values to a file. The object would use the services of a `Scanner` object to read the values from the keyboard, and then use the services of a `PrintWriter` object to write the values to a file. In this example, the object is collaborating with objects created from classes in the Java API. The objects that you create from your own classes can also collaborate with each other.

If one object is to collaborate with another object, then it must know something about the other object's class methods and how to call them. For example, suppose we were to write a class named `StockPurchase`, which uses an object of the `Stock` class (presented earlier in this chapter) to simulate the purchase of a stock. The `StockPurchase` class is responsible for calculating the cost of the stock purchase. To do that, it must know how to call the `Stock` class's `getSharePrice` method to get the price per share of the stock. Code Listing 8-24 shows an example of the `StockPurchase` class. (This file is in the source code folder *Chapter 08\StockPurchase Class*.)

**Code Listing 8-24** (`StockPurchase.java`)

```java
/**
   The StockPurchase class represents a stock purchase.
*/

public class StockPurchase
{
   private Stock stock; // The stock that was purchased
   private int shares; // Number of shares owned

   /**
      Constructor
      @param stockObject The stock to purchase.
      @param numShares The number of shares.
   */

   public StockPurchase(Stock stockObject, int numShares)
   {
      // Create a copy of the object referenced by
      // stockObject.
      stock = new Stock(stockObject);
      shares = numShares;
   }

   /**
      getStock method
      @return A copy of the Stock object for the stock
              being purchased.
   */

   public Stock getStock()
   {
      // Return a copy of the object referenced by stock.
      return new Stock(stock);
   }

   /**
      getShares method
      @return The number of shares being purchased.
   */

   public int getShares()
   {
      return shares;
   }

   /**
      getCost method
      @return The cost of the stock purchase.
   */

   public double getCost()
   {
      return shares * stock.getSharePrice();
   }
}
```

The constructor for this class accepts a `Stock` object representing the stock that is being purchased, and an `int` representing the number of shares to purchase. In line 20 we see the first collaboration: The `StockPurchase` constructor makes a copy of the `Stock` object by using the `Stock` class's copy constructor. The copy constructor is used again in the `getStock` method, in line 33, to return a copy of the `Stock` object.

The next collaboration takes place in the `getCost` method. This method calculates and returns the cost of the stock purchase. In line 53 it calls the `Stock` class's `getSharePrice` method to determine the stock's price per share. The program in Code Listing 8-25 demonstrates this class. (This file is also stored in the source code folder *Chapter 08\StockPurchase Class*.)

**Code Listing 8-25** `(StockTrader.java)`

```java
import java.util.Scanner;

/**
   This program allows you to purchase shares of XYZ
   company's stock.
*/

public class StockTrader
{
   public static void main(String[] args)
   {
      int sharesToBuy; // Number of shares to buy.

      // Create a Stock object for the company stock.
      // The trading symbol is XYZ and the stock is
      // currently $9.62 per share.
      Stock xyzCompany = new Stock("XYZ", 9.62);

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Display the current share price.
      System.out.printf("XYZ stock is currently $%,.2f.\n",
                        xyzCompany.getSharePrice());

      // Get the number of shares to purchase.
      System.out.print("How many shares do you want to buy? ");
      sharesToBuy = keyboard.nextInt();

      // Create a StockPurchase object for the transaction.
      StockPurchase buy =
               new StockPurchase(xyzCompany, sharesToBuy);

      // Display the cost of the transaction.
      System.out.printf("Cost of the stock: $%,.2f",
                        buy.getCost());
   }
}
```

**Program Output with Example Input Shown in Bold**

```
XYZ stock is currently $9.62.
How many shares do you want to buy? 100 [Enter]
Cost of the stock: $962.00
```

## Determining Class Collaborations with CRC Cards

During the object-oriented design process, you can determine many of the collaborations that will be necessary among classes by examining the responsibilities of the classes. In Chapter 6, Section 6.9, we discussed the process of finding the classes and their responsibilities. Recall from that section that a class's responsibilities are as follows:

- Things that the class is responsible for knowing
- Actions that the class is responsible for doing

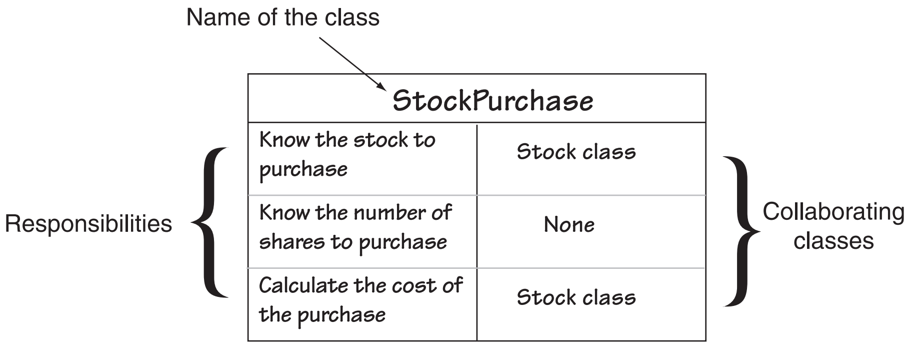
Often you will determine that the class must collaborate with another class in order to fulfill one or more of its responsibilities. One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards. CRC stands for class, responsibilities, and collaborations.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which is discussed in Chapter 6, Section 6.9), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as the following:

- Will an object of this class need to get data from another object in order to fulfill this responsibility?
- Will an object of this class need to request another object to perform an operation in order to fulfill this responsibility?

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. Figure 8-20 shows an example CRC card for the `StockPurchase` class.

**Figure 8-20** CRC card


Name of the class
StockPurchase
Responsibilities
Know the stock to purchase
Stock class
Know the number of shares to purchase
None
Calculate the cost of the purchase
Stock class
Collaborating classes

From the CRC card shown in the figure, we can see that the `StockPurchase` class has the following responsibilities and collaborations:

- **Responsibility:** To know the stock to purchase
  **Collaboration:** The `Stock` class
- **Responsibility:** To know the number of shares to purchase
  **Collaboration:** None
- **Responsibility:** To calculate the cost of the purchase
  **Collaboration:** The `Stock` class

When you have completed a CRC card for each class in the application, you will have a good idea of each class's responsibilities and how the classes must interact.

## 8.12 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Attempting to refer to an instance field or instance method in a static method.** Static methods can refer only to other class members that are static.
- **In a method that accepts an object as an argument, writing code that accidentally modifies the object.** When a reference variable is passed as an argument to a method, the method has access to the object that the variable references. When writing a method that receives a reference variable as an argument, you must take care not to accidentally modify the contents of the object that is referenced by the variable.
- **Allowing a `null` reference to be used.** Because a `null` reference variable does not reference an object, you cannot use it to perform an operation that would require the existence of an object. For example, a `null` reference variable cannot be used to call a method. If you attempt to perform an operation with a `null` reference variable, the program will terminate. This can happen when a class has a reference variable as a field, and it is not properly initialized with the address of an object.
- **Forgetting to use the fully qualified name of an `enum` constant.** Under most circumstances you must use the fully qualified name of an `enum` constant. One exception to this is when the `enum` constant is used as a case expression in a switch statement.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This type of method cannot access any non-static member variables in its own class.
   a. instance
   b. `void`
   c. static
   d. non-static
2. When an object is passed as an argument to a method, this is actually passed.
   a. a copy of the object
   b. the name of the object
   c. a reference to the object
   d. none of these; you cannot pass an object
3. If you write this method for a class, Java will automatically call it any time you concatenate an object of the class with a string.
   a. `toString`
   b. `plusString`
   c. `stringConvert`
   d. `concatString`
4. Making an instance of one class a field in another class is called __________.
   a. nesting
   b. class fielding
   c. aggregation
   d. concatenation
5. This is the name of a reference variable that is always available to an instance method and refers to the object that is calling the method.
   a. `callingObject`
   b. `this`
   c. `me`
   d. `instance`
6. This `enum` method returns the position of an `enum` constant in the declaration.
   a. `position`
   b. `location`
   c. `ordinal`
   d. `toString`
7. Assuming the following declaration exists:

   `enum Seasons { SPRING, WINTER, SUMMER, FALL }`

   what is the fully qualified name of the `FALL` constant?
   a. `FALL`
   b. `enum.FALL`
   c. `FALL.Seasons`
   d. `Seasons.FALL`

8. You cannot use the fully qualified name of an `enum` constant for this.
    a. a `switch` expression
    b. a `case` expression
    c. an argument to a method
    d. all of these
9. The Java Virtual Machine periodically performs this process, which automatically removes unreferenced objects from memory.
    a. memory cleansing
    b. memory deallocation
    c. garbage collection
    d. object expungement
10. If a class has this method, it is called automatically just before an instance of the class is destroyed by the Java Virtual Machine.
    a. `finalize`
    b. `destroy`
    c. `remove`
    d. `housekeeper`
11. CRC stands for
    a. Class, Return value, Composition
    b. Class, Responsibilities, Collaborations
    c. Class, Responsibilities, Composition
    d. Compare, Return, Continue
12. **True or False:** A static member method may refer to non-static member variables of the same class, but only after an instance of the class has been defined.
13. **True or False:** All static member variables are initialized to −1 by default.
14. **True or False:** When an object is passed as an argument to a method, the method can access the argument.
15. **True or False:** A method cannot return a reference to an object.
16. **True or False:** You can declare an enumerated data type inside a method.
17. **True or False:** Enumerated data types are actually special types of classes.
18. **True or False:** `enum` constants have a `toString` method.

## Find the Error

The following class definition has an error. What is it?

1.
```
public class MyClass
{
      private int x;
      private double y;

      public static void setValues(int a, double b)
      {
            x = a;
            y = b;
         }
      }
```

2. Assume the following declaration exists :

   `enum Coffee { MEDIUM, DARK, DECAF }`

   Find the error(s) in the following `switch` statement:

```
// This code has errors!
Coffee myCup = DARK;
switch (myCup)
{
    case Coffee.MEDIUM :
          System.out.println("Mild flavor.");
          break;
    case Coffee.DARK :
          System.out.println("Strong flavor.");
          break;
    case Coffee.DECAF :
          System.out.println("Won't keep you awake.");
          break;
    default:
          System.out.println("Never heard of it.");
}
```

## Algorithm Workbench

1. Consider the following class declaration:

```
public class Circle
{
   private double radius;

   public Circle(double r)
   {
      radius = r;
   }

   public double getArea()
   {
      return Math.PI * radius * radius;
   }

   public double getRadius()
   {
      return radius;
   }
}
```

   a. Write a `toString` method for this class. The method should return a string containing the radius and area of the circle.
   b. Write an `equals` method for this class. The method should accept a `Circle` object as an argument. It should return `true` if the argument object contains the same data as the calling object, or `false` otherwise.
   c. Write a `greaterThan` method for this class. The method should accept a `Circle` object as an argument. It should return `true` if the argument object has an area that is greater than the area of the calling object, or `false` otherwise.

2. Consider the following class declaration:

```
public class Thing
{
     private int x;
     private int y;
     private static int z = 0;

     public Thing()
     {
          x = z;
          y = z;
     }
     static void putThing(int a)
     {
          z = a;
     }
}
```

   Assume a program containing the class declaration defines three `Thing` objects with the following statements:

```
Thing one = new Thing();
Thing two = new Thing();
Thing three = new Thing();
```

   a. How many separate instances of the `x` member exist?
   b. How many separate instances of the `y` member exist?
   c. How many separate instances of the `z` member exist?
   d. What value will be stored in the `x` and `y` members of each object?
   e. Write a statement that will call the `putThing` method.

3. A pet store sells dogs, cats, birds, and hamsters. Write a declaration for an enumerated data type that can represent the types of pets the store sells.

### Short Answer

1. Describe one thing you cannot do with a static method.
2. Why are static methods useful in creating utility classes?
3. Describe the difference in the way variables and class objects are passed as arguments to a method.

4. Even if you do not write an `equals` method for a class, Java provides one. Describe the behavior of the `equals` method that Java automatically provides.
5. A "has a" relationship can exist between classes. What does this mean?
6. What happens if you attempt to call a method using a reference variable that is set to `null`?
7. Is it advisable or not advisable to write a method that returns a reference to an object that is a private field? What is the exception to this?
8. What is the `this` key word?
9. Look at the following declaration:

   ```
   enum Color { RED, ORANGE, GREEN, BLUE }
   ```

   a. What is the name of the data type declared by this statement?
   b. What are the `enum` constants for this type?
   c. Write a statement that defines a variable of this type and initializes it with a valid value.
10. Assuming the following `enum` declaration exists:

    ```
    enum Dog { POODLE, BOXER, TERRIER }
    ```

    what will the following statements display?

    a.
    ```
    System.out.println(Dog.POODLE + "\n" +
                       Dog.BOXER + "\n" +
                       Dog.TERRIER);
    ```
    b.
    ```
    System.out.println(Dog.POODLE.ordinal() + "\n" +
                       Dog.BOXER.ordinal() + "\n" +
                       Dog.TERRIER.ordinal());
    ```
    c.
    ```
    Dog myDog = Dog.BOXER;
    if (myDog.compareTo(Dog.TERRIER) > 0)
        System.out.println(myDog + " is greater than " +
                           Dog.TERRIER);
    else
      System.out.println(myDog + " is NOT greater than " +
                         Dog.TERRIER);
    ```
11. Under what circumstances does an object become a candidate for garbage collection?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Area Class

Write a class that has three overloaded static methods for calculating the areas of the following geometric shapes:

- circles
- rectangles
- cylinders

Here are the formulas for calculating the area of the shapes.

Area of a circle: $Area = \pi r^2$
where $\pi$ is `Math.PI` and $r$ is the circle's radius
Area of a rectangle: $Area = Width \times Length$
Area of a cylinder: $Area = \pi r^2 h$
where $\pi$ is `Math.PI`, $r$ is the radius of the cylinder's base, and $h$ is the cylinder's height

Because the three methods are to be overloaded, they should each have the same name, but different parameter lists. Demonstrate the class in a complete program.

VideoNote
The `BankAccount` Class Copy Constructor Problem

### 2. `BankAccount` Class Copy Constructor

Add a copy constructor to the `BankAccount` class. This constructor should accept a `BankAccount` object as an argument. It should assign to the `balance` field the value in the argument's `balance` field. As a result, the new object will be a copy of the argument object.

### 3. Carpet Calculator

The Westfield Carpet Company has asked you to write an application that calculates the price of carpeting for rectangular rooms. To calculate the price, you multiply the area of the floor (width times length) by the price per square foot of carpet. For example, the area of floor that is 12 feet long and 10 feet wide is 120 square feet. To cover that floor with carpet that costs \$8 per square foot would cost \$960. ($12 \times 10 \times 8 = 960$.)

First, you should create a class named `RoomDimension` that has two fields: one for the length of the room and one for the width. The `RoomDimension` class should have a method that returns the area of the room. (The area of the room is the room's length multiplied by the room's width.)

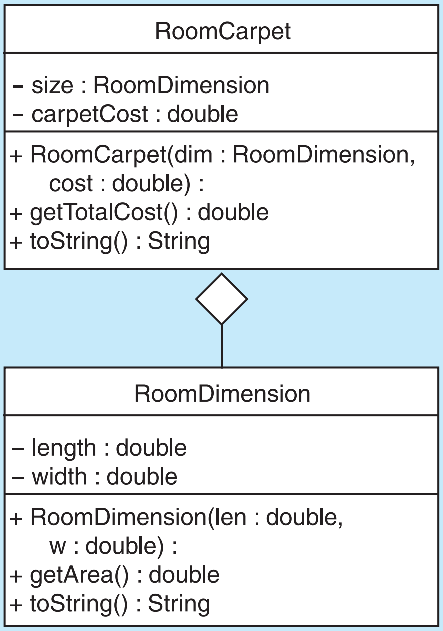
Next you should create a `RoomCarpet` class that has a `RoomDimension` object as a field. It should also have a field for the cost of the carpet per square foot. The `RoomCarpet` class should have a method that returns the total cost of the carpet.

Figure 8-21 is a UML diagram that shows possible class designs and the relationships among the classes. Once you have written these classes, use them in an application that asks the user to enter the dimensions of a room and the price per square foot of the desired carpeting. The application should display the total cost of the carpet.

### 4. `LandTract` Class

Make a `LandTract` class that has two fields: one for the tract's length and one for the width. The class should have a method that returns the tract's area, as well as an `equals` method and a `toString` method. Demonstrate the class in a program that asks the user to enter the dimensions for two tracts of land. The program should display the area of each tract of land and indicate whether the tracts are of equal size.

**Figure 8-21** UML diagram for Programming Challenge 3


RoomCarpet
– size : RoomDimension
– carpetCost : double
+ RoomCarpet(dim : RoomDimension, cost : double) :
+ getTotalCost() : double
+ toString() : String
RoomDimension
– length : double
– width : double
+ RoomDimension(len : double, w : double) :
+ getArea() : double
+ toString() : String

### 5. `Month` Class

Write a class named `Month`. The class should have an `int` field named `monthNumber` that holds the number of the month. For example, January would be 1, February would be 2, and so forth. In addition, provide the following methods:

- A no-arg constructor that sets the `monthNumber` field to 1.
- A constructor that accepts the number of the month as an argument. It should set the `monthNumber` field to the value passed as the argument. If a value less than 1 or greater than 12 is passed, the constructor should set `monthNumber` to 1.
- A constructor that accepts the name of the month, such as "January" or "February" as an argument. It should set the `monthNumber` field to the correct corresponding value.
- A `setMonthNumber` method that accepts an `int` argument, which is assigned to the `monthNumber` field. If a value less than 1 or greater than 12 is passed, the method should set `monthNumber` to 1.
- A `getMonthNumber` method that returns the value in the `monthNumber` field.
- A `getMonthName` method that returns the name of the month. For example, if the `monthNumber` field contains 1, then this method should return "January".
- A `toString` method that returns the same value as the `getMonthName` method.
- An `equals` method that accepts a `Month` object as an argument. If the argument object holds the same data as the calling object, this method should return `true`. Otherwise, it should return `false`.

- A `greaterThan` method that accepts a `Month` object as an argument. If the calling object's `monthNumber` field is greater than the argument's `monthNumber` field, this method should return `true`. Otherwise, it should return `false`.
- A `lessThan` method that accepts a `Month` object as an argument. If the calling object's `monthNumber` field is less than the argument's `monthNumber` field, this method should return `true`. Otherwise, it should return `false`.

### 6. `CashRegister` Class

Write a `CashRegister` class that can be used with the `RetailItem` class that you wrote in Chapter 6's Programming Challenge 4. The `CashRegister` class should simulate the sale of a retail item. It should have a constructor that accepts a `RetailItem` object as an argument. The constructor should also accept an integer that represents the quantity of items being purchased. In addition, the class should have the following methods:

- The `getSubtotal` method should return the subtotal of the sale, which is the quantity multiplied by the price. This method must get the price from the `RetailItem` object that was passed as an argument to the constructor.
- The `getTax` method should return the amount of sales tax on the purchase. The sales tax rate is 6 percent of a retail sale.
- The `getTotal` method should return the total of the sale, which is the subtotal plus the sales tax.

Demonstrate the class in a program that asks the user for the quantity of items being purchased, and then displays the sale's subtotal, amount of sales tax, and total.

### 7. Sales Receipt File

Modify the program you wrote in Programming Challenge 6 to create a file containing a sales receipt. The program should ask the user for the quantity of items being purchased, and then generate a file with contents similar to the following:

```
SALES RECEIPT
Unit Price: $10.00
Quantity: 5
Subtotal: $50.00
Sales Tax: $ 3.00
Total: $53.00
```

### 8. Parking Ticket Simulator

For this assignment you will design a set of classes that work together to simulate a police officer issuing a parking ticket. You should design the following classes:

- **The `ParkedCar` Class:** This class should simulate a parked car. The class's responsibilities are as follows:
  - To know the car's make, model, color, license number, and the number of minutes that the car has been parked.
- **The `ParkingMeter` Class:** This class should simulate a parking meter. The class's only responsibility is as follows:
  - To know the number of minutes of parking time that has been purchased.

- **The `ParkingTicket` Class:** This class should simulate a parking ticket. The class's responsibilities are as follows:
  - To report the make, model, color, and license number of the illegally parked car
  - To report the amount of the fine, which is $25 for the first hour or part of an hour that the car is illegally parked, plus $10 for every additional hour or part of an hour that the car is illegally parked
  - To report the name and badge number of the police officer issuing the ticket
- **The `PoliceOfficer` Class:** This class should simulate a police officer inspecting parked cars. The class's responsibilities are as follows:
  - To know the police officer's name and badge number
  - To examine a `ParkedCar` object and a `ParkingMeter` object, and determine whether the car's time has expired
  - To issue a parking ticket (generate a `ParkingTicket` object) if the car's time has expired

Write a program that demonstrates how these classes collaborate.

### 9. Geometry Calculator

Design a `Geometry` class with the following methods:

- A static method that accepts the radius of a circle and returns the area of the circle. Use the following formula:

  $Area = \pi r^2$

  Use `Math.PI` for $\pi$ and the radius of the circle for $r$.
- A static method that accepts the length and width of a rectangle and returns the area of the rectangle. Use the following formula:

  $Area = Length \times Width$
- A static method that accepts the length of a triangle's base and the triangle's height. The method should return the area of the triangle. Use the following formula:

  $Area = Base \times Height \times 0.5$

The methods should display an error message if negative values are used for the circle's radius, the rectangle's length or width, or the triangle's base or height.

Next, write a program to test the class, which displays the following menu and responds to the user's selection:

```
Geometry Calculator
1. Calculate the Area of a Circle
2. Calculate the Area of a Rectangle
3. Calculate the Area of a Triangle
4. Quit

Enter your choice (1-4):
```

Display an error message if the user enters a number outside the range of 1 through 4 when selecting an item from the menu.

### 10. Car Instrument Simulator

For this assignment, you will design a set of classes that work together to simulate a car's fuel gauge and odometer. The classes you will design are the following:

- **The `FuelGauge` Class:** This class will simulate a fuel gauge. Its responsibilities are as follows:
  - To know the car's current amount of fuel, in gallons.
  - To report the car's current amount of fuel, in gallons.
  - To be able to increment the amount of fuel by 1 gallon. This simulates putting fuel in the car. (The car can hold a maximum of 15 gallons.)
  - To be able to decrement the amount of fuel by 1 gallon, if the amount of fuel is greater than 0 gallons. This simulates burning fuel as the car runs.
- **The `Odometer` Class:** This class will simulate the car's odometer. Its responsibilities are as follows:
  - To know the car's current mileage.
  - To report the car's current mileage.
  - To be able to increment the current mileage by 1 mile. The maximum mileage the odometer can store is 999,999 miles. When this amount is exceeded, the odometer resets the current mileage to 0.
  - To be able to work with a `FuelGauge` object. It should decrease the `FuelGauge` object's current amount of fuel by 1 gallon for every 24 miles traveled. (The car's fuel economy is 24 miles per gallon.)

Demonstrate the classes by creating instances of each. Simulate filling the car up with fuel, and then run a loop that increments the odometer until the car runs out of fuel. During each loop iteration, print the car's current mileage and amount of fuel.

### 11. First to One Game

This game is meant for two or more players. In the game, each player starts out with 50 points, as each player takes a turn rolling the dice; the amount generated by the dice is subtracted from the player's points. The first player with exactly one point remaining wins. If a player's remaining points minus the amount generated by the dice results in a value less than one, then the amount should be added to the player's points. (As an alternative, the game can be played with a set number turns. In this case, the player with the amount of points closest to one, when all rounds have been played, wins.)

Write a program that simulates the game being played by two players. Use the `Die` class that was presented in Chapter 6 to simulate the dice. Write a `Player` class to simulate the players.

### 12. Heads or Tails Game

This game is meant for two or more players. In this game, the players take turns flipping a coin. Before the coin is flipped, players should guess if the coin will land face up or face down. If a player guesses correctly, then that player is awarded a point. If a player guesses incorrectly, then that player will lose a point. The first player to score five points is the winner.

Write a program that simulates the game being played by two players. Use the `Coin` class that you wrote as an assignment in Chapter 6 (Programming Challenge 17) to simulate the coin. Write a `Player` class to simulate the players.

# CHAPTER 9 Text Processing and More about Wrapper Classes

**TOPICS**

9.1 Introduction to Wrapper Classes
9.2 Character Testing and Conversion with the `Character` Class
9.3 More `String` Methods
9.4 The `StringBuilder` Class
9.5 Tokenizing Strings
9.6 Wrapper Classes for the Numeric Data Types
9.7 Focus on Problem Solving: The `TestScoreReader` Class
9.8 Common Errors to Avoid

## 9.1 Introduction to Wrapper Classes

**CONCEPT: Java provides wrapper classes for the primitive data types. The wrapper class for a given primitive type contains not only a value of that type, but also methods that perform operations related to the type.**

Recall from Chapter 2 that the primitive data types are called “primitive” because they are not created from classes. Instead of instantiating objects, you create variables from the primitive data types, and variables do not have attributes or methods. They are designed simply to hold a single value in memory.

Java also provides wrapper classes for all of the primitive data types. A *wrapper class* is a class that is “wrapped around” a primitive data type and allows you to create objects instead of variables. In addition, these wrapper classes provide methods that perform useful operations on primitive values. For example, you have already used the wrapper class “parse” methods to convert strings to primitive values.

Although these wrapper classes can be used to create objects instead of variables, few programmers use them that way. One reason is because the wrapper classes are immutable, which means that once you create an object, you cannot change the object’s value. Another reason is because they are not as easy to use as variables for simple operations. For example, to get the value stored in an object you must call a method, whereas variables can be used directly in assignment statements, used in mathematical operations, passed as arguments to methods, and so forth.

Although it is not normally useful to create objects from the wrapper classes, they do provide static methods that are very useful. We examine several of Java's wrapper classes in this chapter. We begin by looking at the `Character` class, which is the wrapper class for the `char` data type.

## 9.2 Character Testing and Conversion with the `Character` Class

**CONCEPT: The `Character` class is a wrapper class for the `char` data type. It provides numerous methods for testing and converting character data.**

The `Character` class is part of the `java.lang` package, so no `import` statement is necessary to use this class. The class provides several static methods for testing the value of a `char` variable. Some of these methods are listed in Table 9-1. Each of the methods accepts a single `char` argument and returns a `boolean` value.

**Table 9-1** Some static `Character` class methods for testing `char` values

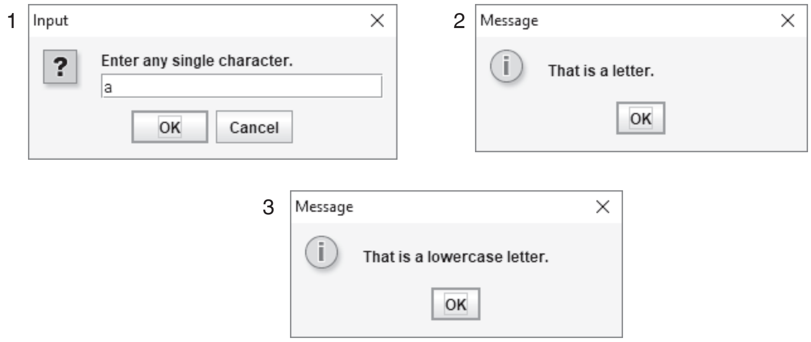
| Method | Description |
|---|---|
| `boolean isDigit(char ch)` | Returns `true` if the argument passed into *ch* is a digit from 0 through 9. Otherwise returns `false`. |
| `boolean isLetter(char ch)` | Returns `true` if the argument passed into *ch* is an alphabetic letter. Otherwise returns `false`. |
| `boolean isLetterOrDigit(char ch)` | Returns `true` if the character passed into *ch* contains a digit (0 through 9) or an alphabetic letter. Otherwise returns `false`. |
| `boolean isLowerCase(char ch)` | Returns `true` if the argument passed into *ch* is a lowercase letter. Otherwise returns `false`. |
| `boolean isUpperCase(char ch)` | Returns `true` if the argument passed into *ch* is an uppercase letter. Otherwise returns `false`. |
| `boolean isSpaceChar(char ch)` | Returns `true` if the argument passed into *ch* is a space character. Otherwise returns `false`. |
| `boolean isWhiteSpace(char ch)` | Returns `true` if the argument passed into *ch* is a whitespace character (a space, tab, or newline character). Otherwise returns `false`. |

The program in Code Listing 9-1 demonstrates many of these methods. Figures 9-1 and 9-2 show example interactions with the program.

**Code Listing 9-1** (CharacterTest.java)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates some of the Character
   class's character testing methods.
*/

public class CharacterTest
{
   public static void main(String[] args)
   {
      String input;  // To hold the user's input
      char ch;       // To hold a single character

      // Get a character from the user and store
      // it in the ch variable.
      input = JOptionPane.showInputDialog("Enter " +
                               "any single character.");
      ch = input.charAt(0);

      // Test the character.
      if (Character.isLetter(ch))
      {
         JOptionPane.showMessageDialog(null,
                           "That is a letter.");
      }

      if (Character.isDigit(ch))
      {
         JOptionPane.showMessageDialog(null,
                           "That is a digit.");
      }

      if (Character.isLowerCase(ch))
      {
         JOptionPane.showMessageDialog(null,
               "That is a lowercase letter.");
      }

      if (Character.isUpperCase(ch))
      {
         JOptionPane.showMessageDialog(null,
               "That is an uppercase letter.");
      }

      if (Character.isSpaceChar(ch))
      {
         JOptionPane.showMessageDialog(null,
                           "That is a space.");
      }

      if (Character.isWhitespace(ch))
      {
         JOptionPane.showMessageDialog(null,
                  "That is a whitespace character.");
      }

      System.exit(0);
  }
}
```

**Figure 9-1** Interaction with the `CharacterTest.java` program (Oracle Corporate Counsel)



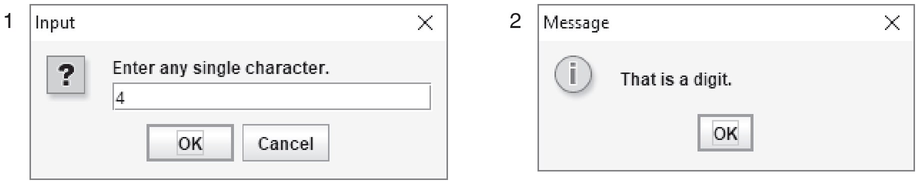
1 Enter any single character.
a

2 That is a letter.

3 That is a lowercase letter.

Code Listing 9-2 shows a more practical application of the character testing methods. It tests a string to determine whether it is a seven-character customer number in the proper format. Figures 9-3 and 9-4 show example interactions with the program.

**Figure 9-2** Interaction with the `CharacterTest.java` program (Oracle Corporate Counsel)



1 Enter any single character.
4

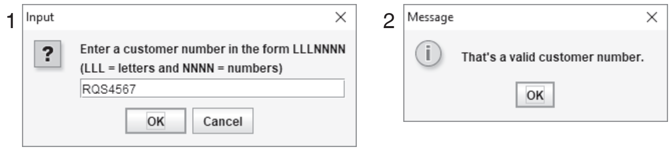
2 That is a digit.

**Code Listing 9-2** (`CustomerNumber.java`)

```
import javax.swing.JOptionPane;

/**
   This program tests a customer number to
   verify that it is in the proper format.
```

```
*/

public class CustomerNumber
{
  public static void main(String[] args)
  {
      String input;  // To hold the user's input

      // Get a customer number.
      input = JOptionPane.showInputDialog("Enter " +
         "a customer number in the form LLLNNNN\n" +
         "(LLL = letters and NNNN = numbers)");

      // Validate the input.
      if (isValid(input))
      {
         JOptionPane.showMessageDialog(null,
                  "That's a valid customer number.");
      }
      else
      {
         JOptionPane.showMessageDialog(null,
             "That is not the proper format of a " +
             "customer number.\nHere is an " +
             "example: ABC1234");
      }

      System.exit(0);
  }

  /**
      The isValid method determines whether a
      String is a valid customer number. If so, it
      returns true.
      @param custNumber The String to test.
      @return true if valid, otherwise false.
  */

  private static boolean isValid(String custNumber)
  {
      boolean goodSoFar = true;   // Flag
      int i = 0;                  // Control variable

      // Test the length.
      if (custNumber.length() != 7)
         goodSoFar = false;

      // Test the first three characters for letters.
      while (goodSoFar && i < 3)
55          {
56             if (!Character.isLetter(custNumber.charAt(i)))
57                goodSoFar = false;
58             i++;
59          }
60
61          // Test the last four characters for digits.
62          while (goodSoFar && i < 7)
63          {
64             if (!Character.isDigit(custNumber.charAt(i)))
65                goodSoFar = false;
66             i++;
67          }
68
69          return goodSoFar;
70       }
71    }
```

**Figure 9-3** Interaction with the `CustomerNumber.java` program (Oracle Corporate Counsel)



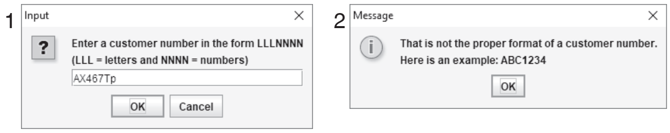
1 Input

Enter a customer number in the form LLLNNNN
(LLL = letters and NNNN = numbers)
RQS4567

2 Message

That's a valid customer number.

**Figure 9-4** Interaction with the `CustomerNumber.java` program (Oracle Corporate Counsel)



1 Input

Enter a customer number in the form LLLNNNN
(LLL = letters and NNNN = numbers)
AX467Tp

2 Message

That is not the proper format of a customer number.
Here is an example: ABC1234

In this program, the customer number is expected to be seven characters long and consist of three alphabetic letters followed by four numeric digits. The `isValid` method accepts a `String` argument, which will be tested. The method uses the following local variables, which are declared in lines 46 and 47:

```
boolean goodSoFar = true;  // Flag
int i = 0;                 // Control variable
```

The `goodSoFar` variable is a flag variable that is initialized with `true`, but will be set to `false` immediately when the method determines the customer number is not in a valid format. The `i` variable is a loop control variable.

The first test is to determine whether the string is the correct length. In line 50 the method tests the length of the `custNumber` argument. If the argument is not seven characters long, it is not valid and the `goodSoFar` variable is set to `false` in line 51.

Next, the method uses the following loop, in lines 54 through 59, to validate the first three characters:

```
while (goodSoFar && i < 3)
{
    if (!Character.isLetter(custNumber.charAt(i)))
       goodSoFar = false;
    i++;
}
```

Recall from Chapter 2 that the `String` class's `charAt` method returns a character at a specific position in a string (position numbering starts at 0). This code uses the `Character.isLetter` method to test the characters at positions 0, 1, and 2 in the `custNumber` string. If any of these characters are not letters, the `goodSoFar` variable is set to `false` and the loop terminates. Next, the method uses the following loop, in lines 62 through 67, to validate the last four characters:

```
while (goodSoFar && i < 7)
{
    if (!Character.isDigit(custNumber.charAt(i)))
       goodSoFar = false;
    i++;
}
```

This code uses the `Character.isDigit` method to test the characters at positions 3, 4, 5, and 6 in the `custNumber` string. If any of these characters are not digits, the `goodSoFar` variable is set to `false` and the loop terminates. Last, the method returns the value of the `goodSoFar` method.

## Character Case Conversion

The `Character` class also provides the static methods listed in Table 9-2 for converting the case of a character. Each method accepts a `char` argument and returns a `char` value.

**Table 9-2** Some `Character` class methods for case conversion

| Method | Description |
|---|---|
| `char toLowerCase(char` *`ch`*`)` | Returns the lowercase equivalent of the argument passed to *`ch`*. |
| `char toUpperCase(char` *`ch`*`)` | Returns the uppercase equivalent of the argument passed to *`ch`*. |

If the `toLowerCase` method's argument is an uppercase character, the method returns the lowercase equivalent. For example, the following statement will display the character `a` on the screen:

```
System.out.println(Character.toLowerCase('A'));
```

If the argument is already lowercase, the `toLowerCase` method returns it unchanged. The following statement also causes the lowercase character `a` to be displayed:

```
System.out.println(Character.toLowerCase('a'));
```

If the `toUpperCase` method's argument is a lowercase character, the method returns the uppercase equivalent. For example, the following statement will display the character `A` on the screen:

```
System.out.println(Character.toUpperCase('a'));
```

If the argument is already uppercase, the `toUpperCase` method returns it unchanged.

Any non-letter argument passed to `toLowerCase` or `toUpperCase` is returned as it is. Each of the following statements displays the method argument without any change:

```
System.out.println(Character.toLowerCase('*'));
System.out.println(Character.toLowerCase('$'));
System.out.println(Character.toUpperCase('&'));
System.out.println(Character.toUpperCase('%'));
```

The program in Code Listing 9-3 demonstrates the `toUpperCase` method in a loop that asks the user to enter Y or N. The program repeats as long as the user enters Y or y in response to the question.

**Code Listing 9-3** `(CircleArea.java)`

```
import java.util.Scanner;

/**
   This program demonstrates the Character
   class's toUpperCase method.
*/

public class CircleArea
{
   public static void main(String[] args)
   {
      double radius; // The circle's radius
      double area;   // The circle's area
      String input;  // To hold a line of input
      char choice;   // To hold a single character

      // Create a Scanner object to read keyboard input.
      Scanner keyboard = new Scanner(System.in);

      do
      {
         // Get the circle's radius.
         System.out.print("Enter the circle's radius: ");
         radius = keyboard.nextDouble();

         // Consume the remaining newline character.
         keyboard.nextLine();

         // Calculate and display the area.
         area = Math.PI * radius * radius;
         System.out.printf("The area is %.2f.\n", area);

         // Repeat this?
         System.out.print("Do you want to do this " +
                          "again? (Y or N) ");
         input = keyboard.nextLine();
         choice = input.charAt(0);

      } while (Character.toUpperCase(choice) == 'Y');
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the circle's radius: 10 [Enter]
The area is 314.16.
Do you want to do this again? (Y or N) y [Enter]
Enter the circle's radius: 15 [Enter]
The area is 706.86.
Do you want to do this again? (Y or N) n [Enter]
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.1 Write a statement that converts the contents of the `char` variable `big` to lowercase. The converted value should be assigned to the variable `little`.

9.2 Write an `if` statement that displays the word "digit" if the `char` variable `ch` contains a numeric digit. Otherwise, it should display "Not a digit."

9.3 What is the output of the following statement?

```
System.out.println(Character.toUpperCase(Character.toLowerCase('A')));
```

9.4 Write a loop that asks the user, "Do you want to repeat the program or quit? (R/Q)". The loop should repeat until the user has entered an R or Q (either uppercase or lowercase).

9.5 What will the following code display?

```
char var = '$';
System.out.println(Character.toUpperCase(var));
```

9.6 Write a loop that counts the number of uppercase characters that appear in the `String` object `str`.

## 9.3 More `String` Methods

**CONCEPT:** **The `String` class provides several methods for searching and working with `String` objects.**

### Searching for Substrings

The `String` class provides several methods that search for a string inside of a string. The term *substring* commonly is used to refer to a string that is part of another string. Table 9-3 summarizes some of these methods. Each of the methods in Table 9-3 returns a `boolean` value indicating whether the string was found.

Let's take a closer look at each of these methods.

#### The `startsWith` and `endsWith` Methods

The `startsWith` method determines whether the calling object's string begins with a specified substring. For example, the following code determines whether the string `"Four score and seven years ago"` begins with `"Four"`. The method returns `true` if the string begins with the specified substring, or `false` otherwise.

```
String str = "Four score and seven years ago";
if (str.startsWith("Four"))
   System.out.println("The string starts with Four.");
else
   System.out.println("The string does not start with Four.");
```

**Table 9-3** `String` methods that search for a substring

| Method | Description |
|---|---|
| `boolean startsWith(String str)` | This method returns `true` if the calling string begins with the string passed into `str`. |
| `boolean endsWith(String str)` | This method returns `true` if the calling string ends with the string passed into `str`. |
| `boolean regionMatches(int start, String str, int start2, int n)` | This method returns `true` if a specified region of the calling string matches a specified region of the string passed into `str`. The `start` parameter indicates the starting position of the region within the calling string. The `start2` parameter indicates the starting position of the region within `str`. The `n` parameter indicates the number of characters in both regions. |
| `boolean regionMatches(Boolean ignoreCase, int start, String str, int start2, int n)` | This overloaded version of the `regionMatches` method has an additional parameter, `ignoreCase`. If `true` is passed into this parameter, the method ignores the case of the calling string and `str` when comparing the regions. If `false` is passed into the *ignoreCase* parameter, the comparison is case-sensitive. |

In the code, the method call `str.startsWith("Four")` returns `true` because the string does begin with `"Four"`. The `startsWith` method performs a case-sensitive comparison, so the method call `str.startsWith("four")` would return `false`.

The `endsWith` method determines whether the calling string ends with a specified substring. For example, the following code determines whether the string `"Four score and seven years ago"` ends with `"ago"`. The method returns `true` if the string does end with the specified substring or `false` otherwise.

```
String str = "Four score and seven years ago";
if (str.endsWith("ago"))
   System.out.println("The string ends with ago.");
else
   System.out.println("The string does not end with ago.");
```

In the code, the method call `str.endsWith("ago")` returns `true` because the string does end with “ago”. The `endsWith` method also performs a case-sensitive comparison, so the method call `str.endsWith("Ago")` would return `false`.

The program in Code Listing 9-4 demonstrates a search algorithm that uses the `startsWith` method. The program searches an array of strings for an element that starts with a specified string.

**Code Listing 9-4** (`PersonSearch.java`)

```
import java.util.Scanner;

/**
   This program uses the startsWith method to search using
   a partial string.
*/

public class PersonSearch
{
   public static void main(String[] args)
   {
      String lookUp; // To hold a lookup string

      // Create an array of names.
      String[] people = { "Cutshaw, Will", "Davis, George",
                          "Davis, Jenny", "Russert, Phil",
                          "Russell, Cindy", "Setzer, Charles",
                          "Smathers, Holly", "Smith, Chris",
                          "Smith, Brad", "Williams, Jean" };

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get a partial name to search for.
      System.out.print("Enter the first few characters of " +
                       "the last name to look up: ");
      lookUp = keyboard.nextLine();

      // Display all of the names that begin with the
      // string entered by the user.
      System.out.println("Here are the names that match:");
      for (String person : people)
      {
         if (person.startsWith(lookUp))
            System.out.println(person);
      }
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the first few characters of the last name to look up: Davis [Enter]
Here are the names that match:
Davis, George
Davis, Jenny
```

**Program Output with Example Input Shown in Bold**

```
Enter the first few characters of the last name to look up: Russ [Enter]
Here are the names that match:
Russert, Phil
Russell, Cindy
```

### The `regionMatches` Methods

The `String` class provides overloaded versions of the `regionMatches` method, which determines whether specified regions of two strings match. The following code demonstrates:

```
String str = "Four score and seven years ago";
String str2 = "Those seven years passed quickly";
if (str.regionMatches(15, str2, 6, 11))
   System.out.println("The regions match.");
else
   System.out.println("The regions do not match.");
```

This code will display “The regions match.” The specified region of the `str` string begins at position 15, and the specified region of the `str2` string begins at position 6. Both regions consist of 11 characters. The specified region in the `str` string is “seven years” and the specified region in the `str2` string is also “seven years”. Because the two regions match, the `regionMatches` method in this code returns `true`. This version of the `regionMatches` method performs a case-sensitive comparison. An overloaded version accepts an additional argument indicating whether to perform a case-insensitive comparison. The following code demonstrates:

```
String str = "Four score and seven years ago";
String str2 = "THOSE SEVEN YEARS PASSED QUICKLY";
```

```
if (str.regionMatches(true, 15, str2, 6, 11))
   System.out.println("The regions match.");
else
   System.out.println("The regions do not match.");
```

This code will also display "The regions match." The first argument passed to this version of the `regionMatches` method can be `true` or `false`, indicating whether a case-insensitive comparison should be performed. In this example, `true` is passed, so case will be ignored when the regions "seven years" and "SEVEN YEARS" are compared.

Each of these methods indicates by a `boolean` return value whether a substring appears within a string. The `String` class also provides methods that not only search for items within a string, but also report the location of those items. Table 9-4 describes overloaded versions of the `indexOf` and `lastIndexOf` methods.

**Table 9-4** `String` methods for getting a character or substring's location

| Method | Description |
|---|---|
| `int indexOf(char ch)` | Searches the calling `String` object for the character passed into *ch*. If the character is found, the position of its first occurrence is returned. Otherwise, −1 is returned. |
| `int indexOf(char ch, int start)` | Searches the calling `String` object for the character passed into *ch*, beginning at the position passed into *start* and going to the end of the string. If the character is found, the position of its first occurrence is returned. Otherwise, −1 is returned. |
| `int indexOf(String str)` | Searches the calling `String` object for the string passed into *str*. If the string is found, the beginning position of its first occurrence is returned. Otherwise, −1 is returned. |
| `int indexOf(String str, int start)` | Searches the calling `String` object for the string passed into *str*. The search begins at the position passed into *start* and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, −1 is returned. |
| `int lastIndexOf(char ch)` | Searches the calling `String` object for the character passed into *ch*. If the character is found, the position of its last occurrence is returned. Otherwise, −1 is returned. |
| `int lastIndexOf(char ch, int start)` | Searches the calling `String` object for the character passed into *ch*, beginning at the position passed into *start*. The search is conducted backward through the string, to position 0. If the character is found, the position of its last occurrence is returned. Otherwise, −1 is returned. |

(*Continued*)

**Table 9-4** `String` methods for getting a character or substring's location (*continued*)

| Method | Description |
|---|---|
| `int lastIndexOf(String str)` | Searches the calling `String` object for the string passed into `str`. If the string is found, the beginning position of its last occurrence is returned. Otherwise, −1 is returned. |
| `int lastIndexOf(String str, int start)` | Searches the calling `String` object for the string passed into `str`, beginning at the position passed into `start`. The search is conducted backward through the string, to position 0. If the string is found, the beginning position of its last occurrence is returned. Otherwise, −1 is returned. |

### Finding Characters with the `indexOf` and `lastIndexOf` Methods

The `indexOf` and `lastIndexOf` methods can search for either a character or a substring within the calling string. If the item being searched for is found, its position is returned. Otherwise `-1` is returned. Here is an example of code using two of the methods to search for a character:

```
String str = "Four score and seven years ago";
int first, last;

first = str.indexOf('r');
last = str.lastIndexOf('r');

System.out.println("The letter r first appears at " +
                   "position " + first);

System.out.println("The letter r last appears at " +
                   "position " + last);
```

This code produces the following output:

```
The letter r first appears at position 3
The letter r last appears at position 24
```

The following code shows another example. It uses a loop to show the positions of each letter ‘r’ in the string.

```
String str = "Four score and seven years ago";
int position;

System.out.println("The letter r appears at the " +
                   "following locations:");
position = str.indexOf('r');
while (position != -1)
{
   System.out.println(position);
   position = str.indexOf('r', position + 1);
}
```

This code will produce the following output:

```
The letter r appears at the following locations:
3
8
24
```

The following code is very similar, but it uses the `lastIndexOf` method and shows the positions in reverse order:

```
String str = "Four score and seven years ago";
int position;

System.out.println("The letter r appears at the " +
                   "following locations.");
position = str.lastIndexOf('r');
while (position != −1)
{
   System.out.println(position);
   position = str.lastIndexOf('r', position − 1);
}
```

This code will produce the following output:

```
The letter r appears at the following locations.
24
8
3
```

### Finding Substrings with the `indexOf` and `lastIndexOf` Methods

The `indexOf` and `lastIndexOf` methods can also search for substrings within a string. The following code shows an example. It displays the starting positions of each occurrence of the word “and” within a string.

```
String str = "and a one and a two and a three";
int position;
System.out.println("The word and appears at the " +
                   "following locations.");
position = str.indexOf("and");
while (position != -1)
{
   System.out.println(position);
   position = str.indexOf("and", position + 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
0
10
20
```

The following code also displays the same results, but in reverse order:

```
String str = "and a one and a two and a three";
int position;

System.out.println("The word and appears at the " +
                   "following locations.");
position = str.lastIndexOf("and");
while (position != -1)
{
   System.out.println(position);
   position = str.lastIndexOf("and", position - 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
20
10
0
```

## Extracting Substrings

The `String` class provides several methods that allow you to retrieve a substring from a string. The methods we will examine are listed in Table 9-5.

**Table 9-5** String methods for extracting substrings

| Method | Description |
|---|---|
| `String substring(int start)` | This method returns a copy of the substring that begins at *start* and goes to the end of the calling object's string. |
| `String substring(int start, int end)` | This method returns a copy of a substring. The argument passed into *start* is the substring's starting position, and the argument passed into *end* is the substring's ending position. The character at the *start* position is included in the substring, but the character at the *end* position is not included. |
| `void getChars(int start, int end, char[] array, int arrayStart)` | This method extracts a substring from the calling object and stores it in a `char` array. The argument passed into *start* is the substring's starting position, and the argument passed into *end* is the substring's ending position. The character at the *start* position is included in the substring, but the character at the *end* position is not included. (The last character in the substring ends at *end* – 1.) The characters in the substring are stored as elements in the array that is passed into the *array* parameter. The *arrayStart* parameter specifies the starting subscript within the array where the characters are to be stored. |
| `char[] toCharArray()` | This method returns all of the characters in the calling object as a `char` array. |

### The `substring` Methods

The `substring` method returns a copy of a substring from the calling object. There are two overloaded versions of this method. The first version accepts an `int` argument that is the starting position of the substring. The method returns a reference to a `String` object containing all of the characters from the starting position to the end of the string. The character at the starting position is part of the substring. Here is an example of the method's use:

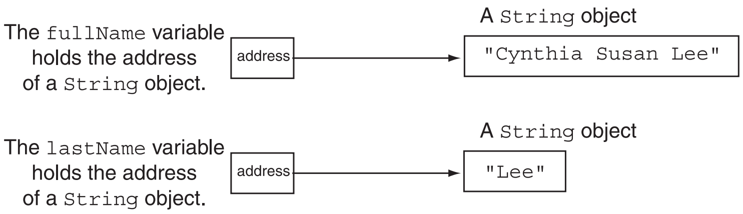
```
String fullName = "Cynthia Susan Lee";
String lastName = fullName.substring(14);
System.out.println("The full name is " + fullName);
System.out.println("The last name is " + lastName);
```

This code will produce the following output:

```
The full name is Cynthia Susan Lee
The last name is Lee
```

Keep in mind that the `substring` method returns a new `String` object that holds a copy of the substring. When this code executes, the `fullName` and `lastName` variables will reference two different `String` objects, as shown in Figure 9-5.

**Figure 9-5** The `fullName` and `lastName` variables reference separate objects


The fullName variable holds the address of a String object.
address
A String object
"Cynthia Susan Lee"
The lastName variable holds the address of a String object.
address
A String object
"Lee"

The second version of the method accepts two `int` arguments. The first specifies the substring's starting position and the second specifies the substring's ending position. The character at the starting position is included in the substring, but the character at the ending position is not. Here is an example of how the method is used:

```
String fullName = "Cynthia Susan Lee";
String middleName = fullName.substring(8, 13);
System.out.println("The full name is " + fullName);
System.out.println("The middle name is " + middleName);
```

The code will produce the following output:

```
The full name is Cynthia Susan Lee
The middle name is Susan
```

### The `getChars` and `toCharArray` Methods

The `getChars` and `toCharArray` methods convert the calling `String` object to a `char` array. The `getChars` method can be used to convert a substring, while the `toCharArray` method converts the entire string. Here is an example of how the `getChars` method might be used:

```
String fullName = "Cynthia Susan Lee";
char[] nameArray = new char[5];
fullName.getChars(8, 13, nameArray, 0);
System.out.println("The full name is " + fullName);
System.out.println("The values in the array are:");
for (int i = 0; i < nameArray.length; i++)
    System.out.print(nameArray[i] + " ");
```

This code stores the individual characters of the substring “Susan” in the elements of the `nameArray` array, beginning at element 0. The code will produce the following output:

```
The full name is Cynthia Susan Lee
The values in the array are:
S u s a n
```

The `toCharArray` method returns a reference to a `char` array that contains all of the characters in the calling object. Here is an example:

```
String fullName = "Cynthia Susan Lee";
char[] nameArray;
nameArray = fullName.toCharArray();
System.out.println("The full name is " + fullName);
System.out.println("The values in the array are:");
for (int i = 0; i < nameArray.length; i++)
    System.out.print(nameArray[i] + " ");
```

This code will produce the following output:

```
The full name is Cynthia Susan Lee
The values in the array are:
C y n t h i a   S u s a n   L e e
```

These methods can be used when you want to use an array processing algorithm on the contents of a `String` object. The program in Code Listing 9-5 converts a `String` object to an array and then uses the array to determine the number of letters, digits, and whitespace characters in the string.

## Code Listing 9-5 (StringAnalyzer.java)

```
import java.util.Scanner;

/**
   This program displays the number of letters,
   digits, and whitespace characters in a string.
*/
```

```

public class StringAnalyzer
{
   public static void main(String [] args)
   {
      String input;        // To hold input
      char[] array;        // Array for input
      int letters = 0;     // Number of letters
      int digits = 0;      // Number of digits
      int whitespaces = 0; // Number of whitespaces

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Prompt the user to enter a line of input.
      System.out.print("Enter a string: ");
      input = keyboard.nextLine();

      // Convert the string to a char array.
      array = input.toCharArray();

      // Analyze the characters.
      for (int i = 0; i < array.length; i++)
      {
         if (Character.isLetter(array[i]))
            letters++;
         else if (Character.isDigit(array[i]))
            digits++;
         else if (Character.isWhitespace(array[i]))
            whitespaces++;
      }

      // Display the results.
      System.out.println("That string contains " +
                         letters + " letters, " +
                         digits + " digits, and " +
                         whitespaces +
                         " whitespace characters.");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter a string: 99 red balloons [Enter]
That string contains 11 letters, 2 digits, and 2 whitespace characters.
```

## Methods That Return a Modified `String`

The `String` class methods listed in Table 9-6 return a modified copy of a `String` object.

**Table 9-6** Methods that return a modified copy of a `String` object

| Method | Description |
|---|---|
| `String concat(String str)` | This method returns a copy of the calling `String` object with the contents of *`str`* concatenated to it. |
| `String replace(char oldChar, char newChar)` | This method returns a copy of the calling `String` object, in which all occurrences of the character passed into *`oldChar`* have been replaced by the character passed into *`newChar`*. |
| `String trim()` | This method returns a copy of the calling `String` object, in which all leading and trailing whitespace characters have been deleted. |

The `concat` method performs the same operation as the `+` operator when used with strings. For example, look at the following code, which uses the `+` operator:

```
String fullName;
String firstName = "Timothy ";
String lastName = "Haynes";
fullName = firstName + lastName;
```

Equivalent code can also be written with the `concat` method. Here is an example:

```
String fullName;
String firstName = "Timothy ";
String lastName = "Haynes";
fullName = firstName.concat(lastName);
```

The `replace` method returns a copy of a `String` object, where every occurrence of a specified character has been replaced with another character. For example, look at the following code:

```
String str1 = "Tom Talbert Tried Trains";
String str2;
str2 = str1.replace('T', 'D');
System.out.println(str1);
System.out.println(str2);
```

In this code, the `replace` method will return a copy of the `str1` object with every occurrence of the letter 'T' replaced with the letter 'D'. The code will produce the following output:

```
Tom Talbert Tried Trains
Dom Dalbert Dried Drains
```

Remember that the `replace` method does not modify the contents of the calling `String` object, but returns a reference to a `String` that is a modified copy of it. After the previous code executes, the `str1` and `str2` variables will reference different `String` objects.

The `trim` method returns a copy of a `String` object with all leading and trailing whitespace characters deleted. A *leading* whitespace character is one that appears at the beginning, or left side, of a string. For example, the following string has three leading whitespace characters:

```
"   Hello"
```

A *trailing* whitespace character is one that appears at the end, or right side, of a string, after the non-space characters. For example, the following string has three trailing whitespace characters:

```
"Hello   "
```

Here is an example:

```
String greeting1 = "   Hello   ";
String greeting2;
greeting2 = greeting1.trim();
System.out.println("*" + greeting1 + "*");
System.out.println("*" + greeting2 + "*");
```

In this code, the first statement assigns the string `"   Hello   "` (with three leading spaces and three trailing spaces) to the `greeting1` variable. The `trim` method is called, which returns a copy of the string with the leading and trailing spaces removed. The code will produce the following output:

```
*   Hello   *
*Hello*
```

One common use of the `trim` method is to remove any leading or trailing spaces that the user might have entered while inputting data.

## The Static `valueOf` Methods

The `String` class has several overloaded versions of a method named `valueOf`. This method accepts a value of any primitive data type as its argument and returns a string representation of the value. Table 9-7 describes these methods.

The following code demonstrates several of these methods:

```
boolean b = true;
char [] letters = { 'a', 'b', 'c', 'd', 'e' };
double d = 2.4981567;
int i = 7;

System.out.println(String.valueOf(b));
System.out.println(String.valueOf(letters));
System.out.println(String.valueOf(letters, 1, 3));
System.out.println(String.valueOf(d));
System.out.println(String.valueOf(i));
```

**Table 9-7** Some of the `String` class's `valueOf` methods

| Method | Description |
|---|---|
| `String valueOf(boolean` *`b`*`)` | If the `boolean` argument passed to *`b`* is `true`, the method returns the string `"true"`. If the argument is `false`, the method returns the string `"false"`. |
| `String valueOf(char` *`c`*`)` | This method returns a `String` containing the character passed into *`c`*. |
| `String valueOf(char[]` *`array`*`)` | This method returns a `String` that contains all of the elements in the `char` array passed into *`array`*. |
| `String valueOf(char[]` *`array`*`, int` *`subscript`*`, int` *`count`*`)` | This method returns a `String` that contains part of the elements in the `char` array passed into *`array`*. The argument passed into *`subscript`* is the starting subscript and the argument passed into *`count`* is the number of elements. |
| `String valueOf(double` *`number`*`)` | This method returns the `String` representation of the `double` argument passed into *`number`*. |
| `String valueOf(float` *`number`*`)` | This method returns the `String` representation of the `float` argument passed into *`number`*. |
| `String valueOf(int` *`number`*`)` | This method returns the `String` representation of the `int` argument passed into *`number`*. |
| `String valueOf(long` *`number`*`)` | This method returns the `String` representation of the `long` argument passed into *`number`*. |

This code will produce the following output:

```
true
abcde
bcd
2.4981567
7
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.7 Write a method that accepts a reference to a `String` object as an argument and returns `true` if the argument ends with the substring "ger". Otherwise, the method should return `false`.

9.8 Modify the method you wrote for Checkpoint 9.7 so it performs a case-insensitive test. The method should return `true` if the argument ends with "ger" in any possible combination of uppercase and lowercase letters.

9.9 Look at the following declaration:

```
String cafeName = "Broadway Cafe";
String str;
```

Which of the following methods would you use to make `str` reference the string "Broadway"?

```
startsWith
regionMatches
substring
indexOf
```

9.10 What is the difference between the `indexOf` and `lastIndexOf` methods?

9.11 What is the difference between the `getChars` and `substring` methods?

9.12 The `+` operator, when used with strings, performs the same operation as what `String` method?

9.13 What is the difference between the `getChars` and `toCharArray` methods?

9.14 Look at the following code:

```
String str1 = "To be, or not to be";
String str2 = str1.replace('o', 'u');
System.out.println(str1);
System.out.println(str2);
```

You hear a fellow student claim that the code will display the following:

```
Tu be ur nut tu be
Tu be ur nut tu be
```

Is your fellow student right or wrong? Why?

9.15 What will the following code display?

```
String str1 = "William ";
String str2 = " the ";
String str3 = " Conqueror";
System.out.println(str1.trim() + str2.trim() +
                   str3.trim());
```

9.16 Assume that a program has the following declarations:

```
double number = 9.47;
String str;
```

Write a statement that assigns a string representation of the `number` variable to `str`.

## 9.4 The StringBuilder Class

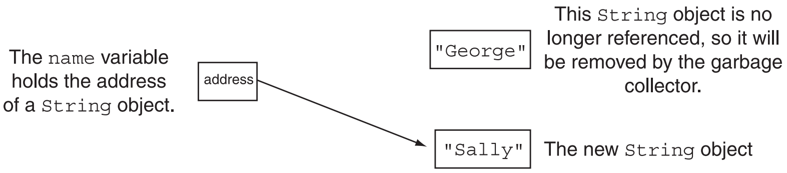
**CONCEPT: The `StringBuilder` class is similar to the `String` class, except that you may change the contents of `StringBuilder` objects. The `StringBuilder` class also provides several useful methods that the `String` class does not have.**

The `StringBuilder` class is similar to the `String` class. The main difference between the two is that you can change the contents of a `StringBuilder` object, but you cannot change the contents of a `String` object. Recall that `String` objects are immutable. This means that once you set the contents of a `String` object, you cannot change the string value that it holds. For example, look at the following code:

```
String name;
name = "George";
name = "Sally";
```

The first statement creates the `name` variable. The second creates a `String` object containing the string "George" and assigns its address to the `name` variable. Although we cannot change the contents of the `String` object, we can make the `name` variable reference a different `String` object. That's what the third statement does: It creates another `String` object containing the string "Sally", and assigns its address to `name`. This is illustrated by Figure 9-6.

**Figure 9-6** The `String` object containing "George" is no longer referenced


The name variable holds the address of a String object.
address
"George"
This String object is no longer referenced, so it will be removed by the garbage collector.
"Sally"
The new String object

Unlike `String` objects, `StringBuilder` objects have methods that allow you to modify their contents without creating a new object in memory. You can change specific characters, insert characters, delete characters, and perform other operations. The `StringBuilder` object will grow or shrink in size, as needed, to accommodate the changes.

The fact that `String` objects are immutable is rarely a problem, but you might consider using `StringBuilder` objects if your program needs to make a lot of changes to one or more strings. This will improve the program's efficiency by reducing the number of `String` objects that must be created and then removed by the garbage collector. Now let's look at the `StringBuilder` class's constructors and methods.

## The `StringBuilder` Constructors

Table 9-8 lists three of the `StringBuilder` constructors.

**Table 9-8** `StringBuilder` constructors

| Constructor | Description |
|---|---|
| `StringBuilder()` | This constructor accepts no arguments. It gives the object enough storage space to hold 16 characters, but no characters are stored in it. |
| `StringBuilder(int` *`length`*`)` | This constructor gives the object enough storage space to hold *`length`* characters, but no characters are stored in it. |
| `StringBuilder(String` *`str`*`)` | This constructor initializes the object with the string in `str`. The object's initial storage space will be the length of the string plus 16. |

The first two constructors create empty `StringBuilder` objects of a specified size. The first constructor makes the `StringBuilder` object large enough to hold 16 characters, and the second constructor makes the object large enough to hold *`length`* characters. Remember, `StringBuilder` objects automatically resize themselves, so it is not a problem if you later want to store a larger string in the object. The third constructor accepts a `String` object as its argument and assigns the object's contents to the `StringBuilder` object. Here is an example of its use:

```
StringBuilder city = new StringBuilder("Charleston");
System.out.println(city);
```

This code creates a `StringBuilder` object and assigns its address to the `city` variable. The object is initialized with the string "Charleston". As demonstrated by this code, you can pass a `StringBuilder` object to the `println` and `print` methods.

One limitation of the `StringBuilder` class is that you cannot use the assignment operator to assign strings to `StringBuilder` objects. For example, the following code will not work:

```
StringBuilder city = "Charleston";  // ERROR!!! Will not work!
```

Instead of using the assignment operator, you must use the `new` key word and a constructor, or one of the `StringBuilder` methods, to store a string in a `StringBuilder` object.

## Other `StringBuilder` Methods

The `StringBuilder` class provides many of the same methods as the `String` class. Table 9-9 lists several of the `StringBuilder` methods that work exactly like their `String` class counterparts.

**Table 9-9** Methods that are common to the `String` and `StringBuilder` classes

```
char charAt(int  position)
void getChars(int  start, int  end, char[]  array, int  arrayStart)
int indexOf(String  str)
int indexOf(String  str, int  start)
int lastIndexOf(String  str)
int lastIndexOf(String  str, int  start)
int length()
String substring(int  start)
String substring(int  start, int  end)
```

In addition, the `StringBuilder` class provides several methods that the `String` class does not have. Let's look at a few of them.

### The append Methods

The `StringBuilder` class has several overloaded versions of a method named `append`. These methods accept an argument, which may be of any primitive data type, a `char` array, or a

`String` object. They append a string representation of their argument to the calling object's current contents. Because there are so many overloaded versions of `append`, we will examine the general form of a typical call to the method as follows:

```
object.append(item);
```

After the method is called, a string representation of *`item`* will be appended to *`object`*'s contents. The following code shows some of the `append` methods being used:

```
StringBuilder str = new StringBuilder();

// Append values to the object.
str.append("We sold ");           // Append a String object.
str.append(12);                   // Append an int.
str.append(" doughnuts for $");   // Append another String.
str.append(15.95);                // Append a double.

// Display the object's contents.
System.out.println(str);
```

This code will produce the following output:

```
We sold 12 doughnuts for $15.95
```

### The `insert` Methods

The `StringBuilder` class also has several overloaded versions of a method named `insert`, which inserts a value into the calling object's string. These methods accept two arguments: an `int` that specifies the position in the calling object's string where the insertion should begin, and the value to be inserted. The value to be inserted may be of any primitive data type, a `char` array, or a `String` object. Because there are so many overloaded versions of `insert`, we will examine the general form of a typical call to the method as follows:

```
object.insert(start, item);
```

In the general form, *`start`* is the starting position of the insertion and *`item`* is the item to be inserted. The following code shows an example:

```
StringBuilder str = new StringBuilder("New City");
str.insert(4, "York ");
System.out.println(str);
```

The first statement creates a `StringBuilder` object initialized with the string `"New City"`. The second statement inserts the string `"York "` into the `StringBuilder` object, beginning at position 4. The characters that are currently in the object beginning at position 4 are moved to the right. In memory, the `StringBuilder` object is automatically expanded in size to accommodate the inserted characters. If these statements were in a complete program and we ran it, we would see `New York City` displayed on the screen.

The following code shows how a `char` array can be inserted into a `StringBuilder` object:

```
char cArray[] =  { '2', '0', ' ' };
StringBuilder str = new StringBuilder("In July we sold cars.");
str.insert(16, cArray);
System.out.println(str);
```

The first statement declares a `char` array named `cArray`, containing the characters `'2'`, `'0'`, and `' '`. The second statement creates a `StringBuilder` object initialized with the string `"In July we sold cars."` The third statement inserts the characters in `cArray` into the `StringBuilder` object, beginning at position 16. The characters that are currently in the object beginning at position 16 are moved to the right. If these statements were in a complete program and we ran it, we would see `In July we sold 20 cars.` displayed on the screen.

### The `replace` Method

The `StringBuilder` class has a `replace` method that differs slightly from the `String` class's `replace` method. While the `String` class's `replace` method replaces the occurrences of one character with another character, the `StringBuilder` class's `replace` method replaces a specified substring with a string. Here is the general form of a call to the method:

```
object.replace(start, end, str);
```

In the general form, *start* is an `int` that specifies the starting position of a substring in the calling object, and *end* is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.) The `str` parameter is a `String` object. After the method executes, the substring will be replaced with `str`. Here is an example:

```
StringBuilder str =
     new StringBuilder("We moved from Chicago to Atlanta.");
str.replace(14, 21, "New York");
System.out.println(str);
```

The `replace` method in this code replaces the word "Chicago" with "New York". The code will produce the following output:

```
We moved from New York to Atlanta.
```

### The `delete`, `deleteCharAt`, and `setCharAt` Methods

The `delete` and `deleteCharAt` methods are used to delete a substring or a character from a `StringBuilder` object. The `setCharAt` method changes a specified character to another value. Table 9-10 describes these methods.

**Table 9-10** The `StringBuilder` class's `delete`, `deleteCharAt`, and `setCharAt` methods

| Method | Description |
|---|---|
| `StringBuilder delete(int start, int end)` | The *start* parameter is an `int` that specifies the starting position of a substring in the calling object, and the *end* parameter is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.) The method will delete the substring. |
| `StringBuilder deleteCharAt (int position)` | The *position* parameter specifies the location of a character that will be deleted. |
| `void setCharAt(int position, char ch)` | This method changes the character at *position* to the value passed into `ch`. |

The following code demonstrates all three of these methods:

```
StringBuilder str =
      new StringBuilder("I ate 100 blueberries!");

// Display the StringBuilder object.
System.out.println(str);

// Delete the '0'.
str.deleteCharAt(8);

// Delete "blue".
str.delete(9, 13);

// Display the StringBuilder object.
System.out.println(str);

// Change the '1' to '5'
str.setCharAt(6, '5');

// Display the StringBuilder object.
System.out.println(str);
```

This code will produce the following output:

```
I ate 100 blueberries!
I ate 10 berries!
I ate 50 berries!
```

## The `toString` Method

If you need to convert a `StringBuilder` object to a regular string, you can call the object's `toString` method. The following code shows an example of a `StringBuilder` object's contents being assigned to a `String` variable:

```
StringBuilder strb = new StringBuilder("This is a test.");
String str = strb.toString();
```

## In the Spotlight:

### Formatting and Unformatting Telephone Numbers

Telephone numbers in the United States are commonly formatted to appear in the following manner:

```
(XXX)XXX-XXXX
```

In the format, `X` represents a digit. The three digits that appear inside the parentheses are the area code. The three digits following the area code are the prefix, and the four digits after the hyphen are the line number. Here is an example:

```
(919)555-1212
```

Although the parentheses and the hyphen make the number easier for people to read, those characters are unnecessary for processing by a computer. In a computer system, a telephone number is commonly stored as an unformatted series of digits, as shown here:

```
9195551212
```

A program that works with telephone numbers usually needs to unformat numbers that have been entered by the user. This means that the parentheses and the hyphen must be removed before the number is stored in a file or processed in some other way. In addition, such a program needs the ability to format the digits so that the number contains the parentheses and the hyphen when it appears on the screen or is printed on paper.

Code Listing 9-6 shows a class named `Telephone` that contains the following static methods:

- `isFormatted`—This method accepts a `String` argument and returns `true` if the argument is formatted as `(XXX)XXX-XXXX`. If the argument is not formatted this way, the method returns `false`.
- `unformat`—This method accepts a `String` argument. If the argument is formatted as `(XXX)XXX-XXXX`, the method returns an unformatted version of the argument with the parentheses and the hyphen removed. Otherwise, the method returns the original argument.
- `format`—This method's purpose is to format a sequence of digits as `(XXX)XXX-XXXX`. The sequence of digits is passed as a `String` argument. If the argument is 10 characters in length, then the method returns the argument with parentheses and a hyphen inserted. Otherwise, the method returns the original argument.

The program in Code Listing 9-7 demonstrates the `Telephone` class.

**Code Listing 9-6** `(Telephone.java)`

```
/**
   The Telephone class provides static methods
   for formatting and unformatting U.S. telephone
   numbers.
*/

public class Telephone
{
   // These constant fields hold the valid lengths of
   // strings that are formatted and unformatted.
   public final static int FORMATTED_LENGTH = 13;
   public final static int UNFORMATTED_LENGTH = 10;

   /**
      The isFormatted method determines whether a
      string is properly formatted as a U.S. telephone
      number in the following manner:
      (XXX)XXX-XXXX
      @param str The string to test.
      @return true if the string is properly formatted,
              or false otherwise.
   */

   public static boolean isFormatted(String str)
   {
      boolean valid;   // Flag to indicate valid format

      // Determine whether str is properly formatted.
      if (str.length() == FORMATTED_LENGTH &&
          str.charAt(0) == '(' &&
          str.charAt(4) == ')' &&
          str.charAt(8) == '-')
          valid = true;
      else
         valid = false;

      // Return the value of the valid flag.
      return valid;
   }

   /**
      The unformat method accepts a string containing
      a telephone number formatted as:
      (XXX)XXX-XXXX.
      If the argument is formatted in this way, the
      method returns an unformatted string where the
      parentheses and hyphen have been removed. Otherwise,
      it returns the original argument.
      @param str The string to unformat.
      @return An unformatted string.
   */

   public static String unformat(String str)
   {
      // Create a StringBuilder initialized with str.
      StringBuilder strb = new StringBuilder(str);

      // If the argument is properly formatted, then
      // unformat it.
      if (isFormatted(str))
      {
         // First, delete the left paren at position 0.
         strb.deleteCharAt(0);

         // Next, delete the right paren. Because of the
         // previous deletion it is now located at
         // position 3.
         strb.deleteCharAt(3);

         // Next, delete the hyphen. Because of the
         // previous deletions it is now located at
         // position 6.
         strb.deleteCharAt(6);
     }

      // Return the unformatted string.
      return strb.toString();
   }

   /**
      The format method formats a string as:
      (XXX)XXX-XXXX.
      If the length of the argument is UNFORMATTED_LENGTH
      the method returns the formatted string. Otherwise,
      it returns the original argument.
      @param str The string to format.
      @return A string formatted as a U.S. telephone number.
   */

   public static String format(String str)
   {
      // Create a StringBuilder initialized with str.
      StringBuilder strb = new StringBuilder(str);

      // If the argument is the correct length, then
      // format it.
      if (str.length() == UNFORMATTED_LENGTH)
      {
         // First, insert the left paren at position 0.
         strb.insert(0, "(");

         // Next, insert the right paren at position 4.
         strb.insert(4, ")");

         // Next, insert the hyphen at position 8.
         strb.insert(8, "-");
      }

      // Return the formatted string.
      return strb.toString();
   }
}
```

**Code Listing 9-7** `(TelephoneTester.java)`

```java
import java.util.Scanner;

/**
   This program demonstrates the Telephone
   class's static methods.
*/

public class TelephoneTester
{
   public static void main(String[] args)
   {
      String phoneNumber;  // To hold a phone number

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get an unformatted telephone number.
      System.out.print("Enter an unformatted telephone number: ");
      phoneNumber = keyboard.nextLine();

      // Format the telephone number.
      System.out.println("Formatted: " +
             Telephone.format(phoneNumber));

      // Get a formatted telephone number.
      System.out.println("Enter a telephone number formatted as");
      System.out.print("(XXX)XXX-XXXX : ");
      phoneNumber = keyboard.nextLine();

      // Unformat the telephone number.
      System.out.println("Unformatted: " +
             Telephone.unformat(phoneNumber));
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter an unformatted telephone number: 9195551212 [Enter]
Formatted: (919)555-1212
Enter a telephone number formatted as
(XXX)XXX-XXXX: (828)555-1212 [Enter]
Unformatted: 8285551212
```

**NOTE:** The Java API provides a class named `StringBuffer`, which is essentially the same as the `StringBuilder` class, with the same constructors and the same methods. The difference is that the methods in the `StringBuffer` class are *synchronized*. This means that the `StringBuffer` class is safe to use in a multithreaded application. Multithreaded programming is beyond the scope of this book, but in a nutshell, a *multithreaded application* is one that concurrently runs multiple threads of execution. In such an application, it is possible for more than one thread to access the same objects in memory at the same time. In multithreaded applications, it is important that the methods be synchronized, to prevent the possibility of data corruption.

Because synchronization requires extra steps to be performed, the `StringBuffer` class is slower than the `StringBuilder` class. In an application where the object will not be accessed by multiple threads, you should use the `StringBuilder` class to get the best performance. In an application where multiple threads will be accessing the object, you should use the `StringBuffer` class to ensure that its data does not become corrupted.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

9.17 The `String` class is immutable. What does this mean?

9.18 In a program that makes lots of changes to strings, would it be more efficient to use `String` objects or `StringBuilder` objects? Why?

9.19 Look at the following statement:

```
String city = "Asheville";
```

Rewrite this statement so that `city` references a `StringBuilder` object instead of a `String` object.

9.20 You wish to add a string to the end of the existing contents of a `StringBuilder` object. What method do you use?

9.21 You wish to insert a string into the existing contents of a `StringBuilder` object. What method do you use?

9.22 You wish to delete a specific character from the existing contents of a `StringBuilder` object. What method do you use?

9.23 You wish to change a specific character in a `StringBuilder` object. What method do you use?

9.24 How does the `StringBuilder` class's `replace` method differ from the `String` class's `replace` method?

## 9.5 Tokenizing Strings

**CONCEPT: Tokenizing a string is a process of breaking a string down into its components, which are called tokens. The `String` class's `split` method can be used to tokenize strings.**

Sometimes a string will contain a series of words or other items of data separated by spaces or other characters. For example, look at the following string:

```
"peach raspberry strawberry vanilla"
```

This string contains the following four items of data: `peach`, `raspberry`, `strawberry`, and `vanilla`. In programming terms, items such as these are known as *tokens*. Notice that a space appears between the items. The character that separates tokens is known as a *delimiter*. Here is another example:

```
"17;92;81;12;46;5"
```

This string contains the following tokens: `17`, `92`, `81`, `12`, `46`, and `5`. Notice that a semicolon appears between each item. In this example the semicolon is used as a delimiter. Some programming problems require you to read a string that contains a list of items and then extract all of the tokens from the string for processing. For example, look at the following string that contains a date:

```
"3-22-2018"
```

The tokens in this string are `3`, `22`, and `2018`, and the delimiter is the hyphen character. Perhaps a program needs to extract the month, day, and year from such a string. Another example is an operating system pathname, such as the following:

```
/home/rsullivan/data
```

The tokens in this string are `home`, `rsullivan`, and `data`, and the delimiter is the `/` character. Perhaps a program needs to extract all of the directory names from such a pathname.

The process of breaking a string into tokens is known as *tokenizing*. In this section, we will discuss the `String` class's `split` method, a tool that you can use to tokenize strings.

The `String` class has a method named `split`, which tokenizes a string and returns an array of `String` objects. Each element in the array is one of the tokens. As an example, look at Code Listing 9-8.

**Code Listing 9-8**

```
/**
   This program demonstrates the String class's
   split method.
*/

public class SplitDemo1
{
   public static void main(String[] args)
   {
      // Create a string to tokenize.
      String str = "one two three four";

      // Get the tokens, using a space delimiter.
      String[] tokens = str.split(" ");
15
16        // Display the tokens.
17        for (String s : tokens)
18           System.out.println(s);
19     }
20  }
```

**Program Output**

```
one
two
three
four
```

In line 11, we assign the string `"one two three four"` to the `str` variable. Notice that the words are separated by spaces. Line 14 calls the `str` object's `split` method. The argument passed to the `split` method indicates the delimiter. In this example, a space is used as the delimiter. The method returns an array containing the strings `"one"`, `"two"`, `"three"`, and `"four"`. The array is assigned to the `tokens` variable. Then, the `for` loop in lines 17 and 18 displays the elements of the array.

The argument that you pass to the split method is a *regular expression*. A regular expression is a string that specifies a pattern of characters. Regular expressions can be powerful tools, and are commonly used to search for patterns that exist in strings, files, or other collections of text. A complete discussion of regular expressions is outside the scope of this book. However, we will discuss some basic uses of regular expressions for the purpose of tokenizing strings.

In the previous example, we passed a string containing a single space to the `split` method. This specified that the space character was the delimiter. The `split` method also allows you to use multi-character delimiters. This means you are not limited to a single character as a delimiter. Your delimiters can be entire words, if you wish. The following code, which is taken from the program *SplitDemo2.java* in this chapter's source code, demonstrates:

```
// Create a string to tokenize.
String str = "one and two and three and four";

// Get the tokens, using " and " as the delimiter.
String[] tokens = str.split(" and ");

// Display the tokens.
for (String s : tokens)
   System.out.println(s);
```

This code will produce the following output:

```
one
two
three
four
```

The previous code demonstrates multi-character delimiters (delimiters containing multiple characters). You can also specify a series of characters where each individual character is a delimiter. For example, look at the following email address:

```
joe@gaddisbooks.com
```

This string uses two delimiters: `@` (the "at" character) and `.` (the period). To specify that both the `@` character and the `.` character are delimiters, we must enclose them in brackets inside our regular expression. The regular expression will look like this:

```
"[@.]"
```

Because the `@` and `.` characters are enclosed in brackets, they will each be considered as a delimiter. The following code, which is taken from the program *SplitDemo3.java* in this chapter's source code, demonstrates:

```
// Create a string to tokenize.
String str = "joe@gaddisbooks.com";

// Get the tokens, using @ and . as delimiters.
String[] tokens = str.split("[@.]");

// Display the tokens.
for (String s : tokens)
   System.out.println(s);
```

This code will produce the following output:

```
joe
gaddisbooks
com
```

### Trimming a String before Tokenizing

When you are tokenizing a string that was entered by the user, and you are using characters other than whitespaces as delimiters, you will probably want to trim the string before tokenizing it. Otherwise, if the user enters leading whitespace characters, they will become part of the first token. Likewise, if the user enters trailing whitespace characters, they will become part of the last token. For example, look at the following code:

```
// Create a string with leading and trailing whitespaces.
String str = " one;two;three ";

// Tokenize the string using the semicolon as a delimiter.
String[] tokens = str.split(";");

// Display the tokens.
for (String s : tokens)
{
   System.out.println("*" + s + "*");
}
```

Notice that the string referenced by `str` contains a leading and a trailing space. This code will produce the following output:

```
* one*
*two*
*three *
```

Notice in the output that the first token contains the leading space and the last token contains the trailing space. To prevent leading and/or trailing whitespace characters from being included in the tokens, use the `String class's trim` method to remove them. Here is the same code, modified to use the `trim` method:

```
// Create a string with leading and trailing whitespaces.
String str = " one;two;three ";

// Trim leading and trailing whitespace.
str = str.trim();

// Tokenize the string using the semicolon as a delimiter.
String[] tokens = str.split(";");

// Display the tokens.
for (String s : tokens)
{
   System.out.println("*" + s + "*");
}
```

This code will produce the following output:

```
*one*
*two*
*three*
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.25 The following string contains three tokens. What are they? What character is the delimiter?

```
"apples pears bananas"
```

9.26 Look at the following code:

```
String str = "one two three four";
String[] tokens = str.split(" ");
int x = tokens.length;

String first = tokens[0];
```

What value will be stored in `x`? What value will the `first` variable reference?

9.27 Look at the following string:

```
"/home/rjones/mydata.txt"
```

Write code using the `String` class's `split` method that can be used to extract the following tokens from the string: `home` , `rjones` , `mydata` , and `txt`.

9.28 Look at the following string:

```
"dog$cat@bird%squirrel"
```

Write code using the `String` class's `split` method that can be used to extract the following tokens from the string: `dog`, `cat`, `bird`, and `squirrel`.

## 9.6 Wrapper Classes for the Numeric Data Types

**CONCEPT: The Java API provides wrapper classes for each of the numeric data types. These classes have methods that perform useful operations involving primitive numeric values.**

Earlier in this chapter, we discussed the `Character` wrapper class and some of its static methods. The Java API also provides wrapper classes for all of the numeric primitive data types, as listed in Table 9-11.

You have already used many of these wrapper classes' "parse" methods, which convert strings to values of the primitive types. For example, the `Integer.parseInt` method converts a string to an `int`, and the `Double.parseDouble` method converts a string to a `double`. Now we will examine other methods and uses of the wrapper classes.

**Table 9-11** Wrapper classes for the numeric primitive data types

| Wrapper Class | Primitive Type It Applies To |
|---|---|
| `Byte` | `byte` |
| `Double` | `double` |
| `Float` | `float` |
| `Integer` | `int` |
| `Long` | `long` |
| `Short` | `short` |

### The Static `toString` Methods

Each of the numeric wrapper classes has a static `toString` method that converts a number to a string. The method accepts the number as its argument and returns a string representation of that number. The following code demonstrates:

```
int i = 12;
double d = 14.95;
String str1 = Integer.toString(i);
String str2 = Double.toString(d);
```

### The `toBinaryString`, `toHexString`, and `toOctalString` Methods

The `toBinaryString`, `toHexString`, and `toOctalString` methods are static members of the `Integer` and `Long` wrapper classes. These methods accept an integer as an argument and return a string representation of that number converted to binary, hexadecimal, or octal. The following code demonstrates these methods:

```
int number = 14;
System.out.println(Integer.toBinaryString(number));
System.out.println(Integer.toHexString(number));
System.out.println(Integer.toOctalString(number));
```

This code will produce the following output:

```
1110
e
16
```

## The `MIN_VALUE` and `MAX_VALUE` Constants

The numeric wrapper classes each have a set of static `final` variables named `MIN_VALUE` and `MAX_VALUE`. These variables hold the minimum and maximum values for a particular data type. For example, `Integer.MAX_VALUE` holds the maximum value that an `int` can hold. For example, the following code displays the minimum and maximum values for an `int`:

```
System.out.println("The minimum value for an " +
                   "int is " + Integer.MIN_VALUE);
System.out.println("The maximum value for an " +
                   "int is " + Integer.MAX_VALUE);
```

## Autoboxing and Unboxing

It is possible to create objects from the wrapper classes. One way is to pass an initial value to the constructor, as shown here:

```
Integer number = new Integer(7);
```

This creates an `Integer` object initialized with the value 7, referenced by the variable `number`. Another way is to simply declare a wrapper class variable, and then assign a primitive value to it. For example, look at the following code:

```
Integer number;
number = 7;
```

The first statement in this code declares an `Integer` variable named `number`. It does not create an `Integer` object, just a variable. The second statement is a simple assignment statement. It assigns the primitive value 7 to the variable. You might suspect that this will cause an error. After all, `number` is a reference variable, not a primitive variable. However, because `number` is a wrapper class variable, Java performs an autoboxing operation. *Autoboxing* is Java's process of automatically "boxing up" a value inside an object. When this assignment statement executes, Java boxes up the value 7 inside an `Integer` object, and then assigns the address of that object to the `number` variable.

*Unboxing* is the opposite of boxing. It is the process of converting a wrapper class object to a primitive type. The following code demonstrates an unboxing operation:

```
Integer myInt = 5;         // Autoboxes the value 5
int primitiveNumber;
primitiveNumber = myInt;   // Unboxes the object
```

The first statement in this code declares `myInt` as an `Integer` reference variable. The primitive value 5 is autoboxed, and the address of the resulting object is assigned to the `myInt` variable. The second statement declares `primitiveNumber` as an `int` variable. Then, the third statement assigns the `myInt` object to `primitiveNumber`. When this statement executes, Java automatically unboxes the `myInt` wrapper class object and stores the resulting value, which is 5, in `primitiveNumber`.

Although you rarely need to create an instance of a wrapper class, Java's autoboxing and unboxing features make some operations more convenient. Occasionally, you will find yourself in a situation where you want to perform an operation using a primitive variable, but the operation can only be used with an object. For example, recall the `ArrayList` class that we discussed in Chapter 7. An `ArrayList` is an array-like object that can be used to store other objects. You cannot, however, store primitive values in an `ArrayList`. It is intended for objects only. If you try to compile the following statement, an error will occur:

```
ArrayList<int> list = new ArrayList<int>();  // ERROR!
```

However, you can store wrapper class objects in an `ArrayList`. If we need to store `int` values in an `ArrayList`, we have to specify that the `ArrayList` will hold `Integer` objects. Here is an example:

```
ArrayList<Integer> list = new ArrayList<Integer>(); // Okay.
```

This statement declares that `list` references an `ArrayList` that can hold `Integer` objects. One way to store an `int` value in the `ArrayList` is to instantiate an `Integer` object, initialize it with the desired `int` value, and then pass the `Integer` object to the `ArrayList`'s `add` method. Here is an example:

```
ArrayList<Integer> list = new ArrayList<Integer>();
Integer myInt = 5;
list.add(myInt);
```

However, Java's autoboxing and unboxing features make it unnecessary to create the `Integer` object. If you add an `int` value to the `ArrayList`, Java will autobox the value. The following code works without any problems:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
```

When the value 5 is passed to the add method, Java boxes the value up in an `Integer` object. When necessary, Java also unboxes values that are retrieved from the `ArrayList`. The following code demonstrates this:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
int primitiveNumber = list.get(0);
```

The last statement in this code retrieves the item at index 0. Because the item is being assigned to an `int` variable, Java unboxes it and stores the primitive value in the `int` variable.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.29 Write a statement that converts the following integer to a string and stores it in the String object referenced by `str`:

```
int i = 99;
```

9.30 What wrapper class methods convert a number from decimal to another numbering system? What wrapper classes are these methods a member of?

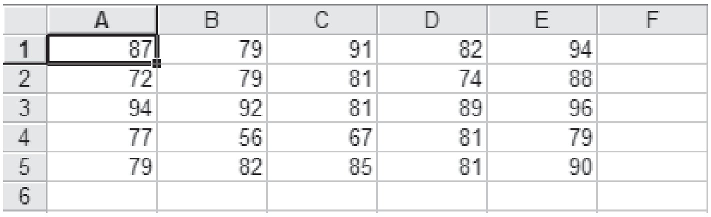
9.31 What is the purpose of the `MIN_VALUE` and `MAX_VALUE` variables that are members of the numeric wrapper classes?

## 9.7

## Focus on Problem Solving: The `TestScoreReader` Class

Professor Harrison keeps her students' test scores in a Microsoft Excel spreadsheet. Figure 9-7 shows a set of five test scores for five students. Each column holds a test score and each row represents the scores for one student.

**Figure 9-7** Microsoft Excel spreadsheet



| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | 87 | 79 | 91 | 82 | 94 | |
| 2 | 72 | 79 | 81 | 74 | 88 | |
| 3 | 94 | 92 | 81 | 89 | 96 | |
| 4 | 77 | 56 | 67 | 81 | 79 | |
| 5 | 79 | 82 | 85 | 81 | 90 | |
| 6 | | | | | | |

In addition to manipulating the scores in Excel, Dr. Harrison wants to write a Java application that accesses them. Excel, like many commercial applications, has the ability to export data to a text file. When the data in a spreadsheet is exported, each row is written to a line, and the values in the cells are separated by commas. For example, when the data shown in Figure 9-7 is exported, it will be written to a text file in the following format:

```
87,79,91,82,94
72,79,81,74,88
94,92,81,89,96
77,56,67,81,79
79,82,85,81,90
```

This is called the *comma separated value* file format. When you save a spreadsheet in this format, Excel saves it to a file with the *.csv* extension. Dr. Harrison decides to export her spreadsheet to a *.csv* file, and then write a Java program that reads the file. The program will use the `String` class's `split` method to extract the test scores from each line, and a wrapper class to convert the tokens to numeric values. As an experiment, she writes the `TestScoreReader` class shown in Code Listing 9-9.

**Code Listing 9-9** (`TestScoreReader.java`)

```
import java.io.*;
import java.util.Scanner;

/**
   The TestScoreReader class reads test scores as
   tokens from a file and calculates the average
   of each line of scores.
*/

```

```
public class TestScoreReader
{
   private Scanner inputFile;
   private String line;

   /**
      The constructor opens a file to read
      the grades from.
      @param filename The file to open.
   */

   public TestScoreReader(String filename)
                          throws IOException
   {
      File file = new File(filename);
      inputFile = new Scanner(file);
   }

   /**
      The readNextLine method reads the next line
      from the file.
      @return true if the line was read, false
      otherwise.
   */

   public boolean readNextLine() throws IOException
   {
      boolean lineRead; // Flag variable

      // Determine whether there is more to read.
      lineRead = inputFile.hasNext();

      // If so, read the next line.
      if (lineRead)
        line = inputFile.nextLine();

      return lineRead;
   }

   /**
      The getAverage method calculates the average
      of the last set of test scores read from the file.
      @return The average.
   */

   public double getAverage()
   {
      int total = 0;  // Accumulator
      double average; // The average test score

      // Tokenize the last line read from the file.
      String[] tokens = line.split(",");

      // Calculate the total of the test scores.
      for (String str : tokens)
      {
         total += Integer.parseInt(str);
      }

      // Calculate the average of the scores.
      // Use a cast to avoid integer division.
      average = (double) total / tokens.length;

      // Return the average.
      return average;
   }

   /**
      The close method closes the file.
   */

   public void close() throws IOException
   {
      inputFile.close();
   }
}
```

The constructor accepts the name of a file as an argument and opens the file. The `readNextLine` method reads a line from the file and stores it in the `line` field. The method returns `true` if a line was successfully read from the file, or `false` if there are no more lines to read. The `getAverage` method tokenizes the last line read from the file, converts the tokens to `int` values, and calculates the average of the values. The average is returned. The program in Code Listing 9-10 uses the `TestScoreReader` class to open the file *Grades.csv* and get the averages of the test scores it contains.

**Code Listing 9-10** (`TestAverages.java`)

```
import java.io.*; // Needed for IOException

/**
   This program uses the TestScoreReader class
   to read test scores from a file and get
   their averages.
*/

public class TestAverages
{
   public static void main(String[] args)
                       throws IOException
   {
      double average;         // Test average
      int studentNumber = 1;  // Control variable

      // Create a TestScoreReader object.
      TestScoreReader scoreReader =
                   new TestScoreReader("Grades.csv");

      // Display the averages.
      while (scoreReader.readNextLine())
      {
         // Get the average from the TestScoreReader.
         average = scoreReader.getAverage();

         // Display the student's average.
         System.out.println("Average for student " +
                            studentNumber + " is " +
                            average);

        // Increment the student number.
        studentNumber++;
      }

      // Close the TestScoreReader.
      scoreReader.close();
      System.out.println("No more scores.");
  }
}
```

**Program Output**

```
Average for student 1 is 86.6
Average for student 2 is 78.8
Average for student 3 is 90.4
Average for student 4 is 72.0
Average for student 5 is 83.4
No more scores.
```

Dr. Harrison's class works properly, and she decides that she can expand it to perform other, more complex, operations.

## 9.8 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Using static wrapper class methods as if they were instance methods.** Many of the most useful wrapper class methods are static, and you should call them directly from the class.
- **Trying to use `String` comparison methods such as `startsWith` and `endsWith` for case-insensitive comparisons.** Most of the `String` comparison methods are case-sensitive. Only the `regionMatches` method performs a case-insensitive comparison.
- **Thinking of the first position of a string as 1.** Many of the `String` and `StringBuilder` methods accept a character position within a string as an argument. Remember, the position numbers in a string start at zero. If you think of the first position in a string as 1, you will cause an off-by-one error.
- **Thinking of the ending position of a substring as part of the substring.** Methods such as `getChars` accept the starting and ending position of a substring as arguments. The character at the ***start*** position is included in the substring, but the character at the ***end*** position is not included. (The last character in the substring ends at ***end*** – 1.)
- **Forgetting to trim a string before tokenizing it.** When tokenizing a string, and using characters other than whitespaces as delimiters, you will probably want to trim the string before tokenizing it. Otherwise, if the string contains leading and/or trailing whitespace characters, they will become part of the first and last tokens.

## Review Questions and Exercises

### Multiple Choice and True/False

1. The `isDigit`, `isLetter`, and `isLetterOrDigit` methods are members of this class.
   a. `String`
   b. `Char`
   c. `Character`
   d. `StringBuilder`
2. This method converts a character to uppercase.
   a. `makeUpperCase`
   b. `toUpperCase`
   c. `isUpperCase`
   d. `upperCase`
3. The `startsWith`, `endsWith`, and `regionMatches` methods are members of this class.
   a. `String`
   b. `Char`
   c. `Character`
   d. `Wrapper`

4. The `indexOf` and `lastIndexOf` methods are members of this class.
   a. `String`
   b. `Integer`
   c. `Character`
   d. `Wrapper`
5. The `substring`, `getChars`, and `toCharArray` methods are members of this class.
   a. `String`
   b. `Float`
   c. `Character`
   d. `Wrapper`
6. This `String` class method performs the same operation as the `+` operator when used on strings.
   a. `add`
   b. `join`
   c. `concat`
   d. `plus`
7. The `String` class has several overloaded versions of a method that accepts a value of any primitive data type as its argument and returns a string representation of the value. The name of the method is __________.
   a. `stringValue`
   b. `valueOf`
   c. `getString`
   d. `valToString`
8. If you do not pass an argument to the `StringBuilder` constructor, the object will have enough memory to store this many characters.
   a. 16
   b. 1
   c. 256
   d. Unlimited
9. This is one of the methods that are common to both the `String` and `StringBuilder` classes.
   a. `append`
   b. `insert`
   c. `delete`
   d. `length`
10. To change the value of a specific character in a `StringBuilder` object, use this method.
    a. `changeCharAt`
    b. `setCharAt`
    c. `setChar`
    d. `change`
11. To delete a specific character in a `StringBuilder` object, use this method.
    a. `deleteCharAt`
    b. `removeCharAt`
    c. `removeChar`
    d. `expunge`

12. The character that separates tokens in a string is known as a __________.
    a. separator
    b. tokenizer
    c. delimiter
    d. terminator
13. This `String` method breaks a string into tokens.
    a. `break`
    b. `tokenize`
    c. `getTokens`
    d. `split`
14. These static `final` variables are members of the numeric wrapper classes and hold the minimum and maximum values for a particular data type.
    a. `MIN_VALUE` and `MAX_VALUE`
    b. `MIN` and `MAX`
    c. `MINIMUM` and `MAXIMUM`
    d. `LOWEST` and `HIGHEST`
15. **True or False:** Character testing methods, such as `isLetter`, accept strings as arguments and test each character in the string.
16. **True or False:** If the `toUpperCase` method's argument is already uppercase, it is returned as is, with no changes.
17. **True or False:** If `toLowerCase` method's argument is already lowercase, it will be inadvertently converted to uppercase.
18. **True or False:** The `startsWith` and `endsWith` methods are case-sensitive.
19. **True or False:** There are two versions of the `regionMatches` method: one that is case-sensitive and one that can be case-insensitive.
20. **True or False:** The `indexOf` and `lastIndexOf` methods can find characters, but cannot find substrings.
21. **True or False:** The `String` class's `replace` method can replace individual characters, but cannot replace substrings.
22. **True or False:** The `StringBuilder` class's `replace` method can replace individual characters, but cannot replace substrings.
23. **True or False:** You can use the `=` operator to assign a string to a `StringBuilder` object.

### Find the Error

Find the error in each of the following code segments:

1.
```
int number = 99;
String str;
// Convert number to a string.
str.valueOf(number);
```
2.
```
// Store a name in a StringBuilder object.
StringBuilder name = "Joe Schmoe";
```
3.
```
// Change the very first character of a
// StringBuilder object to 'Z'.
str.setCharAt(1, 'Z');
```

4.
```
// Tokenize a string that is delimited
// with semicolons. The string has 3 tokens.
String str = "one;two;three";
String tokens = str.split(";");
System.out.println(tokens);
```

## Algorithm Workbench

1. The following `if` statement determines whether `choice` is equal to 'Y' or 'y':

   ```
   if (choice == 'Y' || choice == 'y')
   ```

   Rewrite this statement so it makes only one comparison and does not use the `||` operator. (*Hint*: *Use either the* `toUpperCase` *or* `toLowerCase` *method.*)
2. Write a loop that counts the number of space characters that appear in the `String` object `str`.
3. Write a loop that counts the number of digits that appear in the `String` object `str`.
4. Write a loop that counts the number of lowercase characters that appear in the `String` object `str`.
5. Write a method that accepts a reference to a `String` object as an argument and returns `true` if the argument ends with the substring ".com". Otherwise, the method should return `false`.
6. Modify the method you wrote for Algorithm Workbench 5 so it performs a case-insensitive test. The method should return true if the argument ends with ".com" in any possible combination of uppercase and lowercase letters.
7. Write a method that accepts a `StringBuilder` object as an argument and converts all occurrences of the lowercase letter 't' in the object to uppercase.
8. Look at the following string:

   ```
   "cookies>milk>fudge:cake:ice cream"
   ```

   Write code using the `String` class's `split` method that can be used to extract the following tokens from the string: `cookies`, `milk`, `fudge`, `cake`, and `ice cream`.
9. Assume that `d` is a `double` variable. Write an `if` statement that assigns `d` to the `int` variable `i` if the value in `d` is not larger than the maximum value for an `int`.
10. Write code that displays the contents of the `int` variable `i` in binary, hexadecimal, and octal.

## Short Answer

1. Why should you use `StringBuilder` objects instead of `String` objects in a program that makes lots of changes to strings?
2. A program reads a string as input from the user for the purpose of tokenizing it. Why is it a good idea to trim the string before tokenizing it?
3. Each of the numeric wrapper classes has a static `toString` method. What do these methods do?
4. How can you determine the minimum and maximum values that may be stored in a variable of a given data type?

## Programming Challenges

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Backward String

Write a method that accepts a `String` object as an argument and displays its contents backward. For instance, if the string argument is "gravity" the method should display -"ytivarg". Demonstrate the method in a program that asks the user to input a string and then passes it to the method.

### 2. Word Counter

Write a method that accepts a `String` object as an argument and returns the number of words it contains. For instance, if the argument is "Four score and seven years ago" the method should return the number 6. Demonstrate the method in a program that asks the user to input a string and then passes it to the method. The number of words in the string should be displayed on the screen.

### 3. Sentence Capitalizer

VideoNote
The Sentence Capitalizer Problem

Write a method that accepts a `String` object as an argument and returns a copy of the string with the first character of each sentence capitalized. For instance, if the argument is "hello. my name is Joe. what is your name?" the method should return the string "Hello. My name is Joe. What is your name?" Demonstrate the method in a program that asks the user to input a string and then passes it to the method. The modified string should be displayed on the screen.

### 4. Vowels and Consonants

Write a class with a constructor that accepts a `String` object as its argument. The class should have a method that returns the number of vowels in the string, and another method that returns the number of consonants in the string. Demonstrate the class in a program that performs the following steps:

1. The user is asked to enter a string.
2. The program displays the following menu:
    a. Count the number of vowels in the string
    b. Count the number of consonants in the string
    c. Count both the vowels and consonants in the string
    d. Enter another string
    e. Exit the program
3. The program performs the operation selected by the user and repeats until the user selects e, to exit the program.

### 5. Password Verifier

Imagine you are developing a software package for Amazon.com that requires users to enter their own passwords. Your software requires that users' passwords meet the following criteria:

- The password should be at least six characters long.
- The password should contain at least one uppercase and at least one lowercase letter.
- The password should have at least one digit.

Write a class that verifies that a password meets the stated criteria. Demonstrate the class in a program that allows the user to enter a password and then displays a message indicating whether it is valid or not.

### 6. Telemarketing Phone Number List

Write a program that has two parallel arrays of `String` objects. One of the arrays should hold people's names and the other should hold their phone numbers. Here are example contents of both arrays:

| `name` Array Example Contents | `phone` Array Example Contents |
|---|---|
| `"Harrison, Rose"` | `"555-2234"` |
| `"James, Jean"` | `"555-9098"` |
| `"Smith, William"` | `"555-1785"` |
| `"Smith, Brad"` | `"555-9224"` |

The program should ask the user to enter a name or the first few characters of a name to search for in the array. The program should display all of the names that match the user's input and their corresponding phone numbers. For example, if the user enters "Smith", the program should display the following names and phone numbers from the list:

```
Smith, William: 555-1785
Smith, Brad: 555-9224
```

### 7. Check Writer

Write a program that displays a simulated paycheck. The program should ask the user to enter the date, the payee's name, and the amount of the check. It should then display a simulated check with the dollar amount spelled out, as shown here:

```
                                                  Date:     11/24/2012
Pay to the Order of:     John Phillips                      $1920.85
One thousand nine hundred twenty and 85 cents
```

### 8. Sum of Numbers in a String

Write a program that asks the user to enter a series of numbers separated by commas. Here is an example of valid input:

```
7,9,10,2,18,6
```

The program should calculate and display the sum of all the numbers.

### 9. Sum of Digits in a String

Write a program that asks the user to enter a series of single digit numbers with nothing separating them. The program should display the sum of all the single digit numbers in the string. For example, if the user enters 2514, the method should return 12, which is the sum of 2, 5, 1, and 4. The program should also display the highest and lowest digits in the string. (*Hint: Convert the string to an array.*)

### 10. Word Counter

Write a program that asks the user for the name of a file. The program should display the number of words that the file contains.

### 11. Sales Analysis

The file *SalesData.txt*, in this chapter's source code folder, contains the dollar amount of sales that a retail store made each day for a number of weeks. Each line in the file contains seven numbers, which are the sales numbers for one week. The numbers are separated by a comma. The following line is an example from the file:

```
2541.36,2965.88,1965.32,1845.23,7021.11,9652.74,1469.36
```

Write a program that opens the file and processes its contents. The program should display the following:

- The total sales for each week
- The average daily sales for each week
- The total sales for all of the weeks
- The average weekly sales
- The week number that had the highest amount of sales
- The week number that had the lowest amount of sales

### 12. Miscellaneous String Operations

Write a class with the following static methods:

- `WordCount`. This method should accept a reference to a `String` object as an argument and return the number of words contained in the object.
- `arrayToString`. This method accepts a `char` array as an argument and converts it to a `String` object. The method should return a reference to the `String` object.
- `mostFrequent`. This method accepts a reference to a `String` object as an argument and returns the character that occurs the most frequently in the object.
- `replaceSubstring`. This method accepts three references to `String` objects as arguments. Let's call them `string1`, `string2`, and `string3`. It searches `string1` for all occurrences of `string2`. When it finds an occurrence of `string2`, it replaces it with `string3`. For example, suppose the three arguments have the following values:

  `string1`: "the dog jumped over the fence"
  `string2`: "the"
  `string3`: "that"

With these three arguments, the method would return a reference to a `String` object with the value "that dog jumped over that fence".

Demonstrate each of these methods in a complete program.

### 13. Alphabetic Telephone Number Translator

Many companies use telephone numbers like 555-GET-FOOD so the number is easier for their customers to remember. On a standard telephone, the alphabetic letters are mapped to numbers in the following fashion:

A, B, and C = 2
D, E, and F = 3
G, H, and I = 4
J, K, and L = 5
M, N, and O = 6
P, Q, R, and S = 7
T, U, and V = 8
W, X, Y, and Z = 9

Write an application that asks the user to enter a 10-character telephone number in the format `XXX-XXX-XXXX`. The application should display the telephone number with any alphabetic characters that appeared in the original translated to their numeric equivalent. For example, if the user enters 555-GET-FOOD the application should display 555-438-3663.

### 14. Word Separator

Write a program that accepts as input a sentence in which all of the words are run together, but the first character of each word is uppercase. Convert the sentence to a string in which the words are separated by spaces and only the first word starts with an uppercase letter. For example, the string "StopAndSmellTheRoses." would be converted to "Stop and smell the roses."

### 15. Pig Latin

Write a program that reads a sentence as input and converts each word to "Pig Latin". In one version of Pig Latin, you convert a word by removing the first letter, placing that letter at the end of the word, and then appending "ay" to the word. Here is an example:

English: I SLEPT MOST OF THE NIGHT
Pig Latin: IAY LEPTSAY OSTMAY FOAY HETAY IGHTNAY

### 16. Morse Code Converter

Morse code is a code where each letter of the English alphabet, each digit, and various punctuation characters are represented by a series of dots and dashes. Table 9-12 shows part of the code. Write a program that asks the user to enter a string, and then converts that string to Morse code. Use hyphens for dashes and periods for dots.

**Table 9-12** Morse code

| Character | Code | Character | Code | Character | Code | Character | Code |
|---|---|---|---|---|---|---|---|
| space | *space* | 6 | - . . . . | G | --. | Q | --.- |
| comma | --..-- | 7 | -- . . . | H | . . . . | R | .-. |
| period | .-.-.- | 8 | ---.. | I | .. | S | . . . |
| question mark | ..--.. | 9 | ----. | J | .--- | T | - |
| 0 | ----- | A | .- | K | -.- | U | ..- |
| 1 | .---- | B | - . . . | L | .-.. | V | . . . - |
| 2 | ..--- | C | -.-. | M | -- | W | .-- |
| 3 | . . . -- | D | -.. | N | -. | X | -..- |
| 4 | . . . .- | E | . | O | --- | Y | -.-- |
| 5 | . . . .. | F | ..-. | P | .--. | Z | --.. |

### 17. Lottery Statistics

To play the PowerBall lottery, you buy a ticket that has five numbers in the range of 1–69, and a “PowerBall” number in the range of 1–35. (You can pick the numbers yourself, or you can let the ticket machine randomly pick them for you.) Then, on a specified date, a winning set of numbers are randomly selected by a machine. If your first five numbers match the first five winning numbers in any order, and your PowerBall number matches the winning PowerBall number, then you win the jackpot, which is a very large amount of money. If your numbers match only some of the winning numbers, you win a lesser amount, depending on how many of the winning numbers you have matched.

In the student sample programs for this book, you will find a file named *pbnumbers.txt*, containing the winning PowerBall numbers that were selected between February 3, 2010 and May 11, 2016 (the file contains 654 sets of winning numbers). Here is an example of the first few lines of the file’s contents:

```
17 22 36 37 52 24
14 22 52 54 59 04
05 08 29 37 38 34
10 14 30 40 51 01
07 08 19 26 36 15
and so on . . .
```

Each line in the file contains the set of six numbers that were selected on a given date. The numbers are separated by a space, and the last number in each line is the PowerBall number for that day. For example, the first line in the file shows the numbers for February 3, 2010, which are 17, 22, 36, 37, 52, and the PowerBall number 24.

Write one or more programs that work with this file to perform the following:

- Display the 10 most common numbers, ordered by frequency
- Display the 10 least common numbers, ordered by frequency
- Display the 10 most overdue numbers (numbers that haven’t been drawn in a long time), ordered from most overdue to least overdue
- Display the frequency of each number 1–69, and the frequency of each Powerball number 1–26

### 18. Gas Prices

In the student sample program files for this chapter, you will find a text file named *GasPrices.txt*. The file contains the weekly average prices for a gallon of gas in the United States, beginning on April 5, 1993, and ending on August 26, 2013. Here is an example of the first few lines of the file’s contents:

```
04-05-1993:1.068
04-12-1993:1.079
04-19-1993:1.079
04-26-1993:1.086
05-03-1993:1.086
```

*and so on . . .*

Each line in the file contains the average price for a gallon of gas on a specific date. Each line is formatted in the following way:

```
MM-DD-YYYY:Price
```

`MM` is the two-digit month, `DD` is the two-digit day, and `YYYY` is the four-digit year. `Price` is the average price per gallon of gas on the specified date.

For this assignment you are to write one or more programs that read the contents of the file and perform the following calculations:

**Average Price Per Year:** Calculate the average price of gas per year, for each year in the file. (The file's data starts in April, 1993, and it ends in August, 2013. (For the years 1993 and 2013, use only the data that is present for those years.)
**Average Price Per Month:** Calculate the average price for each month in the file.
**Highest and Lowest Prices Per Year:** For each year in the file, determine the date and amount for the lowest price, and the highest price.
**List of Prices, Lowest to Highest:** Generate a text file that lists the dates and prices, sorted from the lowest price to the highest.
**List of Prices, Highest to Lowest:** Generate a text file that lists the dates and prices, sorted from the highest price to the lowest.

You can write one program to perform all of these calculations, or you can write different programs, one for each calculation.

CHAPTER

# 10 Inheritance

## TOPICS

10.1 What Is Inheritance?
10.2 Calling the Superclass Constructor
10.3 Overriding Superclass Methods
10.4 Protected Members
10.5 Chains of Inheritance
10.6 The `Object` Class
10.7 Polymorphism
10.8 Abstract Classes and Abstract Methods
10.9 Interfaces
10.10 Anonymous Inner Classes
10.11 Functional Interfaces and Lambda Expressions
10.12 Common Errors to Avoid

## 10.1 What Is Inheritance?

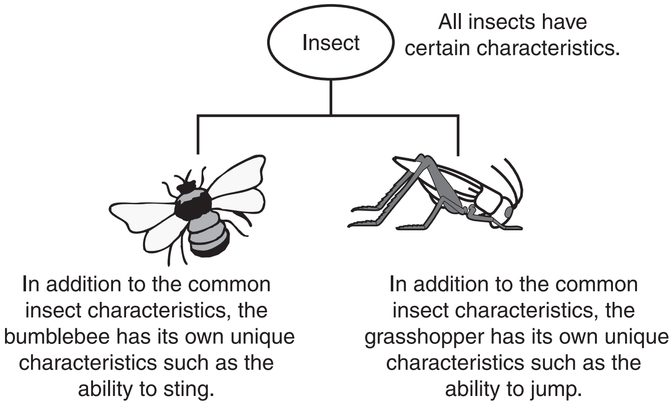
**CONCEPT:** **Inheritance allows a new class to extend an existing class. The new class inherits the members of the class it extends.**

### Generalization and Specialization

VideoNote
Inheritance

In the real world, you can find many objects that are specialized versions of other more general objects. For example, the term *insect* describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 10-1.

**Figure 10-1** Bumblebees and grasshoppers are specialized versions of an insect


Insect
All insects have certain characteristics.
In addition to the common insect characteristics, the bumblebee has its own unique characteristics such as the ability to sting.
In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

## Inheritance and the "Is a" Relationship

When one object is a specialized version of another object, there is an "is a" relationship between them. For example, a grasshopper is an insect. Here are a few other examples of the "is a" relationship:

- A poodle is a dog.
- A car is a vehicle.
- A flower is a plant.
- A rectangle is a shape.
- A football player is an athlete.

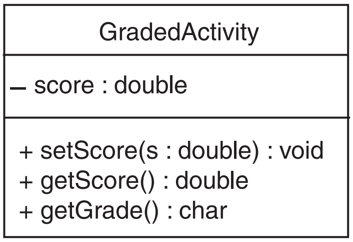
When an "is a" relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, inheritance is used to create an "is a" relationship among classes. This allows you to extend the capabilities of a class by creating another class that is a specialized version of it.

Inheritance involves a superclass and a subclass. The *superclass* is the general class and the *subclass* is the specialized class. You can think of the subclass as an extended version of the superclass. The subclass inherits fields and methods from the superclass without any of them having to be rewritten. Furthermore, new fields and methods may be added to the subclass, and that is what makes it a specialized version of the superclass.

**NOTE:** At the risk of confusing you with too much terminology, it should be mentioned that superclasses are also called *base classes*, and subclasses are also called *derived classes*. Either set of terms is correct. For consistency, this text will use the terms *superclass* and *subclass*.

Let's look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can be given a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D, or F. Figure 10-2 shows a UML diagram for the `GradedActivity` class, which is designed to hold the numeric score of a graded activity. The `setScore` method sets a numeric score, and the `getScore` method returns the numeric score. The `getGrade` method returns the letter grade that corresponds to the numeric score. Notice that the class does not have a programmer-defined constructor, so Java will automatically generate a default constructor for it. This will be a point of discussion later. Code Listing 10-1 shows the code for the class.

**Figure 10-2** UML diagram for the `GradedActivity` class



**Code Listing 10-1** **(GradedActivity.java)**

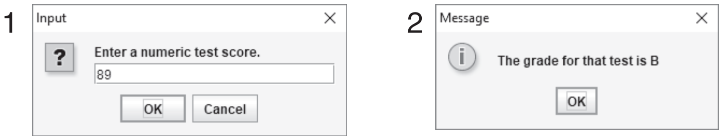
```
/**
   A class that holds a grade for a graded activity.
*/

public class GradedActivity
{
   private double score;   // Numeric score

   /**
      The setScore method sets the score field.
      @param s The value to store in score.
   */

   public void setScore(double s)
   {
      score = s;
   }

   /**
      The getScore method returns the score.
      @return The value stored in the score field.
   */

   public double getScore()
   {
      return score;
   }

   /**
      The getGrade method returns a letter grade
      determined from the score field.
      @return The letter grade.
   */

   public char getGrade()
   {
      char letterGrade;

      if (score >= 90)
         letterGrade = 'A';
      else if (score >= 80)
         letterGrade = 'B';
      else if (score >= 70)
         letterGrade = 'C';
      else if (score >= 60)
         letterGrade = 'D';
      else
         letterGrade = 'F';

      return letterGrade;
   }
}
```

The program in Code Listing 10-2 demonstrates the class. Figures 10-3 and 10-4 show examples of interaction with the program.

**Code Listing 10-2** `(GradeDemo.java)`

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the GradedActivity
   class.
*/

public class GradeDemo
{
   public static void main(String[] args)
   {
        String input;        // To hold input
        double testScore;    // A test score

        // Create a GradedActivity object.
        GradedActivity grade = new GradedActivity();

        // Get a test score.
        input = JOptionPane.showInputDialog("Enter " +
                             "a numeric test score.");
        testScore = Double.parseDouble(input);

        // Store the score in the grade object.
        grade.setScore(testScore);

        // Display the letter grade for the score.
        JOptionPane.showMessageDialog(null,
                       "The grade for that test is " +
                       grade.getGrade());

        System.exit(0);
    }
}
```

**Figure 10-3** Interaction with the `GradeDemo.java` program (Oracle Corporate Counsel)

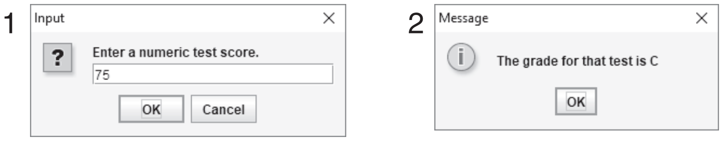


1 Enter a numeric test score.

89

2 The grade for that test is B

**Figure 10-4** Interaction with the `GradeDemo.java` program



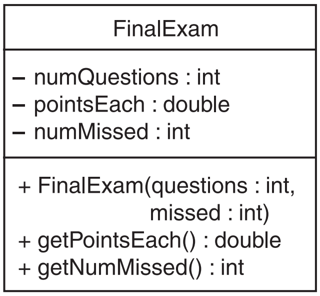
1 Enter a numeric test score.

75

2 The grade for that test is C

The `GradedActivity` class represents the general characteristics of a student's graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create subclasses to handle each one. For example, we could create a `FinalExam` class that would be a subclass of the `GradedActivity` class. Figure 10-5 shows the UML diagram for such a class, and Code Listing 10-3 shows its code. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

**Figure 10-5** UML diagram for the `FinalExam` class

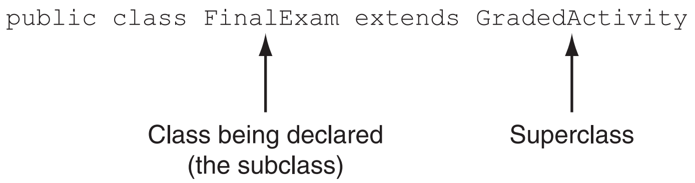

FinalExam
– numQuestions : int
– pointsEach : double
– numMissed : int
+ FinalExam(questions : int, missed : int)
+ getPointsEach() : double
+ getNumMissed() : int

**Code Listing 10-3** (`FinalExam.java`)

```
/**
   This class determines the grade for a final exam.
*/

public class FinalExam extends GradedActivity
{
   private int numQuestions;      // Number of questions
   private double pointsEach;     // Points for each question
   private int numMissed;         // Questions missed

   /**
      The constructor sets the number of questions on the
      exam and the number of questions missed.
      @param questions The number of questions.
      @param missed The number of questions missed.
   */

   public FinalExam(int questions, int missed)
   {
      double numericScore;        // To hold a numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
27          // the numeric score for this exam.
28          pointsEach = 100.0 / questions;
29          numericScore = 100.0 - (missed * pointsEach);
30
31          // Call the inherited setScore method to
32          // set the numeric score.
33          setScore(numericScore);
34       }
35
36       /**
37          The getPointsEach method returns the number of
38          points each question is worth.
39          @return The value in the pointsEach field.
40       */
41
42       public double getPointsEach()
43       {
44          return pointsEach;
45       }
46
47       /**
48          The getNumMissed method returns the number of
49          questions missed.
50          @return The value in the numMissed field.
51       */
52
53       public int getNumMissed()
54       {
55          return numMissed;
56       }
57    }
```

Look at the header for the `FinalExam` class in line 5. The header uses the `extends` key word, which indicates that this class extends another class (a superclass). The name of the superclass is listed after the word `extends`. So, this line indicates that `FinalExam` is the name of the class being declared and `GradedActivity` is the name of the superclass it extends. This is illustrated in Figure 10-6.

**Figure 10-6** `FinalExam` class header



```
public class FinalExam extends GradedActivity
```

If we want to express the relationship between the two classes, we can say that a `FinalExam` is a `GradedActivity`.

Because the `FinalExam` class extends the `GradedActivity` class, it inherits all of the public members of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class.

**Fields:**

| | |
|---|---|
| `int numQuestions;` | Declared in `FinalExam` |
| `double pointsEach;` | Declared in `FinalExam` |
| `int numMissed;` | Declared in `FinalExam` |

**Methods:**

| | |
|---|---|
| Constructor | Declared in `FinalExam` |
| `getPointsEach` | Declared in `FinalExam` |
| `getNumMissed` | Declared in `FinalExam` |
| `setScore` | Inherited from `GradedActivity` |
| `getScore` | Inherited from `GradedActivity` |
| `getGrade` | Inherited from `GradedActivity` |

Notice that the `GradedActivity` class's `score` field is not listed among the members of the `FinalExam` class. That is because the `score` field is private. Private members of the superclass cannot be accessed by the subclass, so technically speaking, they are not inherited. When an object of the subclass is created, the private members of the superclass exist in memory, but only methods in the superclass can access them. They are truly private to the superclass.

You will also notice that the superclass's constructor is not listed among the members of the `FinalExam` class. It makes sense that superclass constructors are not inherited because their purpose is to construct objects of the superclass. In the next section we discuss in more detail how superclass constructors operate.

To see how inheritance works in this example, let's take a closer look at the `FinalExam` constructor in lines 18 through 34. The constructor accepts two arguments: the number of test questions on the exam, and the number of questions missed by the student. In lines 23 and 24 these values are assigned to the `numQuestions` and `numMissed` fields. Then, in lines 28 and 29, the number of points for each question and the numeric test score are calculated. In line 33, the last statement in the constructor reads as follows:

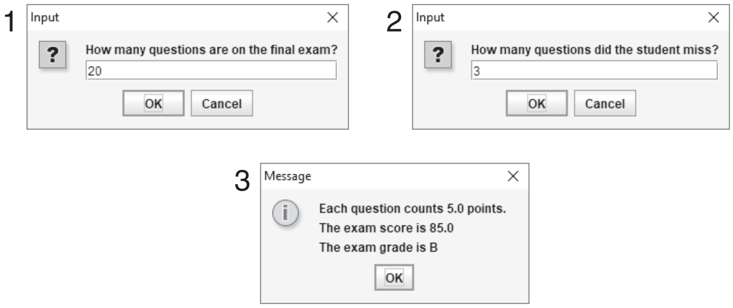
```
setScore(numericScore);
```

This is a call to the `setScore` method. Although no `setScore` method appears in the `FinalExam` class, the method is inherited from the `GradedActivity` class. The program in Code Listing 10-4 demonstrates the `FinalExam` class. Figure 10-7 shows an example of interaction with the program.

**Code Listing 10-4** (`FinalExamDemo.java`)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the FinalExam class,
   which extends the GradedActivity class.
*/

public class FinalExamDemo
{
   public static void main(String[] args)
   {
      String input;        // To hold input
      int questions;       // Number of questions
      int missed;          // Number of questions missed

      // Get the number of questions on the exam.
      input = JOptionPane.showInputDialog("How many " +
                     "questions are on the final exam?");
      questions = Integer.parseInt(input);

      // Get the number of questions the student missed.
      input = JOptionPane.showInputDialog("How many " +
                      "questions did the student miss?");
      missed = Integer.parseInt(input);

      // Create a FinalExam object.
      FinalExam exam = new FinalExam(questions, missed);

      // Display the test results.
      JOptionPane.showMessageDialog(null,
           "Each question counts " + exam.getPointsEach() +
           " points.\nThe exam score is " +
           exam.getScore() + "\nThe exam grade is " +
           exam.getGrade());

      System.exit(0);
   }
}
```

**Figure 10-7** Interaction with the `FinalExamDemo.java` program (Oracle Corporate Counsel)



1 How many questions are on the final exam? 20

2 How many questions did the student miss? 3

3 Each question counts 5.0 points.
The exam score is 85.0
The exam grade is B

In line 27 the following statement creates an instance of the `FinalExam` class and assigns its address to the `exam` variable:

```
FinalExam exam = new FinalExam(questions, missed);
```

When a `FinalExam` object is created in memory, it not only has the members declared in the `FinalExam` class, but also the non-private members declared in the `GradedActivity` class. Notice in lines 30 through 34, shown here, that two public methods of the `GradedActivity` class, `getScore` and `getGrade`, are directly called from the `exam` object:

```
JOptionPane.showMessageDialog(null,
   "Each question counts " + exam.getPointsEach() +
   " points.\nThe exam score is " +
   exam.getScore() + "\nThe exam grade is " +
   exam.getGrade());
```

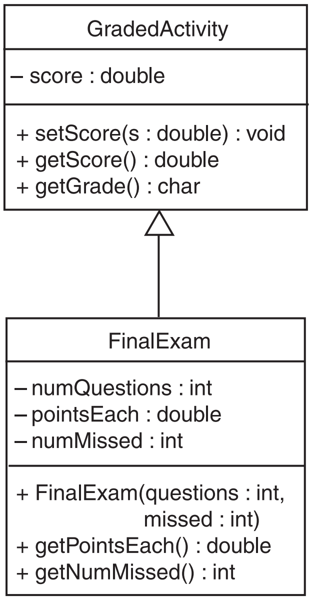
When a subclass extends a superclass, the public members of the superclass become public members of the subclass. In this program the `getScore` and `getGrade` methods can be called from the `exam` object because they are public members of the object's superclass.

As mentioned before, the private members of the superclass (in this case, the `score` field) cannot be accessed by the subclass. When the `exam` object is created in memory, a `score` field exists, but only the methods defined in the superclass, `GradedActivity`, can access it. It is truly private to the superclass. Because the `FinalExam` constructor cannot directly access the `score` field, it must call the superclass's `setScore` method (which is public) to store a value in it.

## Inheritance in UML Diagrams

You show inheritance in a UML diagram by connecting two classes with a line that has an open arrowhead at one end. The arrowhead points to the superclass. Figure 10-8 is a UML diagram showing the relationship between the `GradedActivity` and `FinalExam` classes.

**Figure 10-8** UML diagram showing inheritance


GradedActivity
– score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
FinalExam
– numQuestions : int
– pointsEach : double
– numMissed : int
+ FinalExam(questions : int, missed : int)
+ getPointsEach() : double
+ getNumMissed() : int

## The Superclass's Constructor

You might be wondering how the constructors work together when one class inherits from another. In an inheritance relationship, the superclass constructor always executes before the subclass constructor. As was mentioned earlier, the `GradedActivity` class has only one constructor, which is the default constructor that Java automatically generated for it. When a `FinalExam` object is created, the `GradedActivity` class's default constructor is executed just before the `FinalExam` constructor is executed.

Code Listing 10-5 shows a class, `SuperClass1`, that has a no-arg constructor. The constructor simply displays the message "This is the superclass constructor." Code Listing 10-6 shows `SubClass1`, which extends `SuperClass1`. This class also has a no-arg constructor, which displays the message "This is the subclass constructor."

**Code Listing 10-5** `(SuperClass1.java)`

```
public class SuperClass1
{
   /**
      Constructor
   */

   public SuperClass1()
   {
```

```java
      System.out.println("This is the " +
                  "superclass constructor.");
   }
}
```

**Code Listing 10-6** (`SubClass1.java`)

```java
public class SubClass1 extends SuperClass1
{
   /**
      Constructor
   */

   public SubClass1()
   {
      System.out.println("This is the " +
                  "subclass constructor.");
   }
}
```

The program in Code Listing 10-7 creates a `SubClass1` object. As you can see from the program output, the superclass constructor executes first, followed by the subclass constructor.

**Code Listing 10-7** (`ConstructorDemo1.java`)

```java
/**
   This program demonstrates the order in which
   superclass and subclass constructors are called.
*/

public class ConstructorDemo1
{
   public static void main(String[] args)
   {
      SubClass1 obj = new SubClass1();
   }
}
```

**Program Output**

```
This is the superclass constructor.
This is the subclass constructor.
```

If a superclass has either (a) a default constructor or (b) a no-arg constructor that was written into the class, then that constructor will be automatically called just before a subclass constructor executes. In a moment we will discuss other situations that can arise involving superclass constructors.

## Inheritance Does Not Work in Reverse

In an inheritance relationship, the subclass inherits members from the superclass, not the other way around. This means it is not possible for a superclass to call a subclass's method. For example, if we create a `GradedActivity` object, it cannot call the `getPointsEach` or the `getNumMissed` methods because they are members of the `FinalExam` class.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.1 Here is the first line of a class declaration. What is the name of the superclass? What is the name of the subclass?

```
public class Truck extends Vehicle
```

10.2 Look at the following class declarations and answer the questions that follow them:

```
public class Shape
{
   private double area;
   public void setArea(double a)
   {
      area = a;
   }
   public double getArea()
   {
      return area;
   }
}
public class Circle extends Shape
{
   private double radius;
   public void setRadius(double r)
   {
      radius = r;
      setArea(Math.PI * r * r);
   }
   public double getRadius()
   {
      return radius;
   }
}
```

a) Which class is the superclass? Which class is the subclass?
b) Draw a UML diagram showing the relationship between these two classes.
c) When a `Circle` object is created, what are its public members?
d) What members of the `Shape` class are not accessible to the `Circle` class's methods?
e) Assume a program has the following declarations:

```
Shape s = new Shape();
Circle c = new Circle();
```

Indicate whether the following statements are legal or illegal:

```
c.setRadius(10.0);
s.setRadius(10.0);
System.out.println(c.getArea());
System.out.println(s.getArea());
```

10.3 Class B extends class A. (Class A is the superclass and class B is the subclass.) Describe the order in which the class's constructors execute when a class B object is created.

## 10.2 Calling the Superclass Constructor

**CONCEPT: The `super` key word refers to an object's superclass. You can use the `super` key word to call a superclass constructor.**

In the previous section you saw examples illustrating how a superclass's default constructor or no-arg constructor is automatically called just before the subclass's constructor executes. But what if the superclass does not have a default constructor or a no-arg constructor? Or, what if the superclass has multiple overloaded constructors and you want to make sure a specific one is called? In either of these situations, you use the `super` key word to call a superclass constructor explicitly. The `super` key word refers to an object's superclass and can be used to access members of the superclass.

Code Listing 10-8 shows a class, `SuperClass2`, which has a no-arg constructor and a constructor that accepts an `int` argument. Code Listing 10-9 shows `SubClass2`, which extends `SuperClass2`. This class's constructor uses the `super` key word to call the superclass's constructor and pass an argument to it.

**Code Listing 10-8** (`SuperClass2.java`)

```
public class SuperClass2
{
   /**
      Constructor #1
   */

   public SuperClass2()
   {
      System.out.println("This is the superclass " +
                         "no-arg constructor.");
   }

   /**
      Constructor #2
   */

   public SuperClass2(int arg)
   {
      System.out.println("The following argument " +
                         "was passed to the superclass " +
                         "constructor: " + arg);
   }
}
```

**Code Listing 10-9** (`SubClass2.java`)

```
 1  public class SubClass2 extends SuperClass2
 2  {
 3     /**
 4         Constructor
 5     */
 6
 7     public SubClass2()
 8     {
 9        super(10);
10        System.out.println("This is the " +
11                           "subclass constructor.");
12     }
13  }
```

The statement in line 9 of the `SubClass2` constructor calls the superclass constructor and passes the argument 10 to it. Here are three guidelines you should remember about calling a superclass constructor:

- The `super` statement that calls the superclass constructor may be written only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- The `super` statement that calls the superclass constructor must be the first statement in the subclass's constructor. This is because the superclass's constructor must execute before the code in the subclass's constructor executes.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call the superclass's default constructor, or no-arg constructor, just before

the code in the subclass's constructor executes. This is equivalent to placing the following statement at the beginning of a subclass constructor:

```
super();
```

The program in Code Listing 10-10 demonstrates these classes.

**Code Listing 10-10** (`ConstructorDemo2.java`)

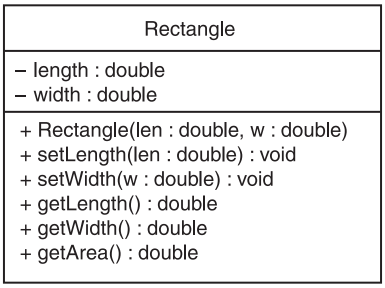
```
/**
   This program demonstrates how a superclass
   constructor is called with the super key word.
*/

public class ConstructorDemo2
{
   public static void main(String[] args)
   {
      SubClass2 obj = new SubClass2();
   }
}
```

**Program Output**

```
The following argument was passed to the superclass constructor: 10
This is the subclass constructor.
```

Let's look at a more meaningful example. Recall the `Rectangle` class from Chapter 6. Figure 10-9 shows a UML diagram for the class.

**Figure 10-9** UML diagram for the `Rectangle` class


Rectangle
– length : double
– width : double
+ Rectangle(len : double, w : double)
+ setLength(len : double) : void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double

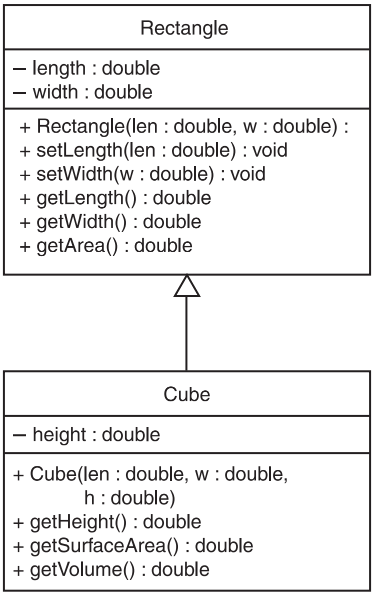
Here is part of the class's code:

```
public class Rectangle
{
   private double length;
   private double width;
   /**
```

```
      Constructor
      @param len The length of the rectangle.
      @param w The width of the rectangle.
   */

   public Rectangle(double len, double w)
   {
      length = len;
      width = w;
   }
   (Other methods follow . . .)
}
```

Next we will design a `Cube` class, which extends the `Rectangle` class. The `Cube` class is designed to hold data about cubes, which not only have a length, width, and area (the area of the base), but also a height, surface area, and volume. A UML diagram showing the inheritance relationship between the `Rectangle` and `Cube` classes is shown in Figure 10-10, and the code for the `Cube` class is shown in Code Listing 10-11.

**Figure 10-10** UML diagram for the `Rectangle` and `Cube` classes


Rectangle
– length : double
– width : double
+ Rectangle(len : double, w : double) :
+ setLength(len : double) : void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double
Cube
– height : double
+ Cube(len : double, w : double, h : double)
+ getHeight() : double
+ getSurfaceArea() : double
+ getVolume() : double

**Code Listing 10-11** (`Cube.java`)

```
/**
   This class holds data about a cube.
*/

public class Cube extends Rectangle
{
   private double height;  // The cube's height

   /**
      The constructor sets the cube's length,
      width, and height.
      @param len The cube's length.
      @param w The cube's width.
      @param h The cube's height.
   */

   public Cube(double len, double w, double h)
   {
      // Call the superclass constructor.
      super(len, w);

      // Set the height.
       height = h;
   }

   /**
      The getHeight method returns the cube's height.
      @return The value in the height field.
   */

   public double getHeight()
   {
      return height;
   }

   /**
      The getSurfaceArea method calculates and
      returns the cube's surface area.
      @return The surface area of the cube.
   */

   public double getSurfaceArea()
   {
      return getArea() * 6;
   }

   /**
      The getVolume method calculates and
      returns the cube's volume.
      @return The volume of the cube.
   */

   public double getVolume()
   {
      return getArea() * height;
   }
}
```

The `Cube` constructor accepts arguments for the parameters `len`, `w`, and `h`. The values that are passed to `len` and `w` are subsequently passed as arguments to the `Rectangle` constructor in line 20:

```
super(len, w);
```

When the `Rectangle` constructor finishes, the remaining code in the `Cube` constructor is executed. The program in Code Listing 10-12 demonstrates the class.

**Code Listing 10-12** (`CubeDemo.java`)

```
import java.util.Scanner;

/**
   This program demonstrates passing arguments to a
   superclass constructor.
*/

public class CubeDemo
{
   public static void main(String[] args)
   {
      double length;       // The cube's length
      double width;        // The cube's width
      double height;       // The cube's height

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get cube's length.
      System.out.println("Enter the following " +
                         "dimensions of a cube:");
      System.out.print("Length: ");
      length = keyboard.nextDouble();

      // Get the cube's width.
      System.out.print("Width: ");
      width = keyboard.nextDouble();

      // Get the cube's height.
      System.out.print("Height: ");
      height = keyboard.nextDouble();

      // Create a cube object and pass the
      // dimensions to the constructor.
      Cube myCube =
             new Cube(length, width, height);

      // Display the cube's properties.
      System.out.println("Here are the cube's " +
                         "properties.");
      System.out.println("Length: " +
                         myCube.getLength());
      System.out.println("Width: " +
                         myCube.getWidth());
      System.out.println("Height: " +
                         myCube.getHeight());
      System.out.println("Base Area: " +
                         myCube.getArea());
      System.out.println("Surface Area: " +
                         myCube.getSurfaceArea());
      System.out.println("Volume: " +
                         myCube.getVolume());
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the following dimensions of a cube:
Length: 10 [Enter]
Width: 15 [Enter]
Height: 12 [Enter]
Here are the cube's properties.
Length: 10.0
Width: 15.0
Height: 12.0
Base Area: 150.0
Surface Area: 900.0
Volume: 1800.0
```

## When the Superclass Has No Default or No-Arg Constructors

Recall from Chapter 6 that Java provides a default constructor for a class only when you provide no constructors for the class. This makes it possible to have a class with no default constructor. The `Rectangle` class we just looked at is an example. It has a constructor that accepts two arguments. Because we have provided this constructor, the `Rectangle` class does not have a default constructor. In addition, we have not written a no-arg constructor for the class.

If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it must call one of the constructors that the superclass does have. If it does not, an error will result when the subclass is compiled.

## Summary of Constructor Issues in Inheritance

We have covered a number of important issues that you should remember about constructors in an inheritance relationship. The following list summarizes them:

- The superclass constructor always executes before the subclass constructor.
- You can write a `super` statement that calls a superclass constructor, but only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- If a `super` statement that calls a superclass constructor appears in a subclass constructor, it must be the first statement.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call `super()` just before the code in the subclass's constructor executes.
- If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it must call one of the constructors that the superclass does have.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.4 Look at the following classes:

```
public class Ground
{
   public Ground()
   {
      System.out.println("You are on the ground.");
   }
}
public class Sky extends Ground
{
   public Sky()
   {
      System.out.println("You are in the sky.");
   }
}
```

What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      Sky object = new Sky();
   }
}
```

10.5 Look at the following classes:

```
public class Ground
{
   public Ground()
   {
       System.out.println("You are on the ground.");
   }
   public Ground(String groundColor)
   {
       System.out.println("The ground is " +
                          groundColor);
   }
}
public class Sky extends Ground
{
   public Sky()
   {
       System.out.println("You are in the sky.");
   }
   public Sky(String skyColor)
   {
       super("green");
       System.out.println("The sky is " + skyColor);
   }
}
```

What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      Sky object = new Sky("blue");
   }
}
```

## 10.3 Overriding Superclass Methods

**CONCEPT: A subclass may have a method with the same signature as a superclass method. In such a case, the subclass method overrides the superclass method.**

Sometimes a subclass inherits a method from its superclass, but the method is inadequate for the subclass's purpose. Because the subclass is more specialized than the superclass, it is sometimes necessary for the subclass to replace inadequate superclass methods with more suitable ones. This is known as method *overriding*.

For example, recall the `GradedActivity` class that was presented earlier in this chapter. This class has a `setScore` method that sets a numeric score and a `getGrade` method that returns a letter grade based on that score. But, suppose a teacher wants to curve a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class she must multiply each student's score by a certain percentage. This gives an adjusted score that is used to determine the letter grade. To satisfy this need we can design a new class, `CurvedActivity`, which extends the `GradedActivity` class and has its own specialized version of the `setScore` method. The `setScore` method in the subclass overrides the `setScore` method in the superclass. Figure 10-11 is a UML diagram showing the relationship between the `GradedActivity` class and the `CurvedActivity` class.

**Figure 10-11** The `GradedActivity` and `CurvedActivity` classes

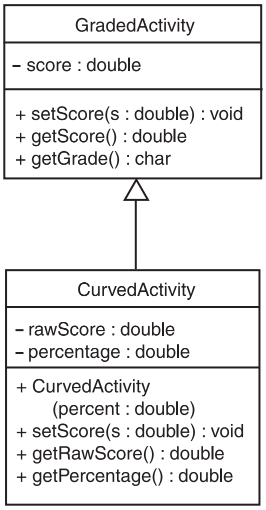

GradedActivity
- score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
CurvedActivity
- rawScore : double
- percentage : double
+ CurvedActivity
(percent : double)
+ setScore(s : double) : void
+ getRawScore() : double
+ getPercentage() : double

Table 10-1 summarizes the `CurvedActivity` class's fields, and Table 10-2 summarizes the class's methods.

**Table 10-1** `CurvedActivity` class fields

| Field | Description |
|---|---|
| `rawScore` | This field holds the student's unadjusted score. |
| `percentage` | This field holds the value that the unadjusted score must be multiplied by to get the curved score. |

**Table 10-2** `CurvedActivity` class methods

| Method | Description |
|---|---|
| `Constructor` | The constructor accepts a `double` argument that is the curve percentage. This value is assigned to the `percentage` field and the `rawScore` field is assigned 0.0. |
| `setScore` | This method overrides the `setScore` method in the superclass. It accepts a `double` argument that is the student's unadjusted score. The method stores the argument in the `rawScore` field, and then passes the result of `rawScore * percentage` as an argument to the superclass's `setScore` method. |
| `getRawScore` | This method returns the value in the `rawScore` field. |
| `getPercentage` | This method returns the value in the `percentage` field. |

Code Listing 10-13 shows the `CurvedActivity` class. The `setScore` method appears in lines 31 through 35. It is important to note that the `setScore` method in the `CurvedActivity` class has the same signature as the `setScore` method in the superclass, `GradedActivity`. In order for overriding to occur, the subclass method must have the same signature as the superclass method. When an object of the subclass invokes the `setScore` method, it invokes the subclass's version of the method, not the superclass's.

**NOTE:** Recall from Chapter 6 that a method's signature consists of the method's name and the data types of the method's parameters, in the order that they appear.

**Code Listing 10-13** `(CurvedActivity.java)`

```
 1 /**
 2    This class computes a curved grade. It extends
 3    the GradedActivity class.
 4 */
 5
 6 public class CurvedActivity extends GradedActivity
 7 {
 8    double rawScore;     // Unadjusted score
 9    double percentage;   // Curve percentage
10
11    /**
12       The constructor sets the curve percentage.
13       @param percent The curve percentage.
14    */
15
16    public CurvedActivity(double percent)
17    {
18       percentage = percent;
19       rawScore = 0.0;
20    }
21
22    /**
23       The setScore method overrides the superclass setScore method.
24       This version accepts the unadjusted score as an argument. That
25       score is multiplied by the curve percentage and the result is
26       sent as an argument to the superclass's setScore method.
27       @param s The unadjusted score.
28    */
29 
30    @Override
31    public void setScore(double s)
32    {
33       rawScore = s;
34       super.setScore(rawScore * percentage);
35    }
36 
37    /**
38       The getRawScore method returns the raw score.
39       @return The value in the rawScore field.
40    */
41 
42    public double getRawScore()
43    {
44       return rawScore;
45    }
46 
47    /**
48       The getPercentage method returns the curve
49       percentage.
50       @return The value in the percentage field.
51    */
52 
53    public double getPercentage()
54    {
55       return percentage;
56    }
57 }
```

Notice that in line 30, the `@Override` annotation appears just before the `setScore` method definition. This annotation tells the Java compiler that the `setScore` method is meant to override a method in the superclass.

The `@Override` annotation in line 30 is not required, but it is recommended that you use it. If the method fails to correctly override a method in the superclass, the compiler will display an error message. For example, suppose we had written the method header in line 31 like this:

```
public void setscore(double s)
```

If you look closely at the method name, you will see that all the letters are written in lowercase. This does not match the method's name in the superclass, which is `setScore`. Without the `@Override` annotation, the code would still compile and execute, but we would not get the expected results because the method in the subclass would not override the method in the superclass. However, by using the `@Override` annotation in line 30, the compiler would generate an error letting us know that the subclass method does not override any method in the superclass.

Let's take a closer look at the `setScore` method in the `CurvedActivity` class. It accepts an argument, which is the student's unadjusted numeric score. This value is stored in the `rawScore` field. Then, in line 34, the following statement is executed:

```
super.setScore(rawScore * percentage);
```

As you already know, the `super` key word refers to the object's superclass. This statement calls the superclass's version of the `setScore` method with the result of the expression `rawScore * percentage` passed as an argument. This is necessary because the superclass's `score` field is private, and the subclass cannot access it directly. In order to store a value in the superclass's `score` field, the subclass must call the superclass's `setScore` method. A subclass may call an overridden superclass method by prefixing its name with the `super` key word and a dot (`.`). The program in Code Listing 10-14 demonstrates this class.

**Code Listing 10-14** (`CurvedActivityDemo.java`)

```
import java.util.Scanner;

/**
   This program demonstrates the CurvedActivity class,
   which inherits from the GradedActivity class.
*/

public class CurvedActivityDemo
{
   public static void main(String[] args)
   {
      double score;              // Raw score
      double curvePercent;       // Curve percentage

      // Create a Scanner object to read keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the unadjusted exam score.
      System.out.print("Enter the student's " +
                       "raw numeric score: ");
      score = keyboard.nextDouble();

      // Get the curve percentage.
      System.out.print("Enter the curve percentage: ");
      curvePercent = keyboard.nextDouble();

      // Create a CurvedActivity object.
      CurvedActivity curvedExam =
                 new CurvedActivity(curvePercent);
30
31       // Set the exam score.
32       curvedExam.setScore(score);
33
34       // Display the raw score.
35       System.out.println("The raw score is " +
36                          curvedExam.getRawScore() +
37                          " points.");
38
39       // Display the curved score.
40       System.out.println("The curved score is " +
41                          curvedExam.getScore());
42
43       // Display the exam grade.
44       System.out.println("The exam grade is " +
45                          curvedExam.getGrade());
46    }
47 }
```

**Program Output with Example Input Shown in Bold**

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage: 1.06 [Enter]
The raw score is 87.0 points.
The curved score is 92.22
The exam grade is A
```

This program uses the `curvedExam` variable to reference a `CurvedActivity` object. In line 32 the following statement is used to call the `setScore` method:

```
curvedExam.setScore(score);
```

Because `curvedExam` references a `CurvedActivity` object, this statement calls the `CurvedActivity` class's `setScore` method, not the superclass's version.

Even though a subclass may override a method in the superclass, superclass objects still call the superclass version of the method. For example, the following code creates an object of the `GradedActivity` class and calls the `setScore` method:

```
GradedActivity regularExam = new GradedActivity();
regularExam.setScore(85);
```

Because `regularExam` references a `GradedActivity` object, this code calls the `GradedActivity` class's version of the `setScore` method.

## Overloading versus Overriding

There is a distinction between overloading a method and overriding a method. Recall from Chapter 6 that overloading is when a method has the same name as one or more other methods, but a different parameter list. Although overloaded methods have the same name, they have different signatures. When a method overrides another method, however, they both have the same signature.

Both overloading and overriding can take place in an inheritance relationship. You already know that overloaded methods can appear within the same class. In addition, a method in a subclass can overload a method in the superclass. If class `A` is the superclass and class `B` is the subclass, a method in class `B` may overload a method in class `A`, or another method in class `B`. Overriding, on the other hand, can only take place in an inheritance relationship. If class `A` is the superclass and class `B` is the subclass, a method in class `B` may override a method in class `A`. However, a method cannot override another method in the same class. The following list summarizes the distinction between overloading and overriding:

- If two methods have the same name but different signatures, they are overloaded. This is true where the methods are in the same class or where one method is in the superclass and the other method is in the subclass.
- If a method in a subclass has the same signature as a method in the superclass, the subclass method overrides the superclass method.

The distinction between overloading and overriding is important because it can affect the accessibility of superclass methods in a subclass. When a subclass overloads a superclass method, both methods may be called with a subclass object. However, when a subclass overrides a superclass method, only the subclass's version of the method can be called with a subclass object. For example, look at the `SuperClass3` class in Code Listing 10-15. It has two overloaded methods named `showValue`. One of the methods accepts an `int` argument and the other accepts a `String` argument.

**Code Listing 10-15** (`SuperClass3.java`)

```
public class SuperClass3
{
   /**
      This method displays an int.
      @param arg An int.
   */

   public void showValue(int arg)
   {
      System.out.println("SUPERCLASS: " +
                         "The int argument was " + arg);
   }

   /**
      This method displays a String.
      @param arg A String.
   */

   public void showValue(String arg)
   {
      System.out.println("SUPERCLASS: " +
                         "The String argument was " + arg);
   }
}
```

Now look at the `SubClass3` class in Code Listing 10-16. It inherits from the `SuperClass3` class.

**Code Listing 10-16** (`SubClass3.java`)

```
 1 public class SubClass3 extends SuperClass3
 2 {
 3    /**
 4       This method overrides one of the superclass methods.
 5       @param arg An int.
 6    */
 7
 8    @Override
 9    public void showValue(int arg)
10    {
11       System.out.println("SUBCLASS: " +
12             "The int argument was " + arg);
13    }
14
15    /**
16       This method overloads the superclass methods.
17       @param arg A double.
18    */
19
20    public void showValue(double arg)
21    {
22       System.out.println("SUBCLASS: " +
23             "The double argument was " + arg);
24    }
25 }
```

Notice that `SubClass3` also has two methods named `showValue`. The first one, in lines 9 through 13, accepts an `int` argument. This method overrides one of the superclass methods because they have the same signature. The second `showValue` method, in lines 20 through 24, accepts a `double` argument. This method overloads the other `showValue` methods because none of the others have the same signature. Although there is a total of four `showValue` methods in these classes, only three of them may be called from a `SubClass3` object. This is demonstrated in Code Listing 10-17.

**Code Listing 10-17** (`ShowValueDemo.java`)

```
/**
   This program demonstrates the methods in the
   SuperClass3 and SubClass3 classes.
*/

public class ShowValueDemo
{
   public static void main(String[] args)
   {
      // Create a SubClass3 object.
      SubClass3 myObject = new SubClass3();

      myObject.showValue(10);          // Pass an int.
      myObject.showValue(1.2);         // Pass a double.
      myObject.showValue("Hello");     // Pass a String.
   }
}
```

**Program Output**

```
SUBCLASS: The int argument was 10
SUBCLASS: The double argument was 1.2
SUPERCLASS: The String argument was Hello
```

When an `int` argument is passed to `showValue`, the subclass's method is called because it overrides the superclass method. In order to call the overridden superclass method, we would have to use the `super` key word in the subclass method. Here is an example:

```
public void showValue(int arg)
{
   super.showValue(arg);   // Call the superclass method.
   System.out.println("SUBCLASS: The int argument was " +
                      arg);
}
```

## Preventing a Method from Being Overridden

When a method is declared with the `final` modifier, it cannot be overridden in a subclass. The following method header is an example that uses the `final` modifier:

```
public final void message()
```

If a subclass attempts to override a `final` method, the compiler generates an error. This technique can be used to make sure that a particular superclass method is used by subclasses and not a modified version of it.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

10.6 Under what circumstances would a subclass need to override a superclass method?

10.7 How can a subclass method call an overridden superclass method?

10.8 If a method in a subclass has the same signature as a method in the superclass, does the subclass method overload or override the superclass method?

10.9 If a method in a subclass has the same name as a method in the superclass, but uses a different parameter list, does the subclass method overload or override the superclass method?

10.10 How do you prevent a method from being overridden?

## 10.4 Protected Members

**CONCEPT: Protected members of a class may be accessed by methods in a subclass, and by methods in the same package as the class.**

Until now you have used two access specifications within a class: `private` and `public`. Java provides a third access specification, `protected`. A protected member of a class may be directly accessed by methods of the same class or methods of a subclass. In addition, protected members may be accessed by methods of any class that are in the same package as the protected member's class. A protected member is not quite private, because it may be accessed by some methods outside the class. Protected members are not quite public either because access to them is restricted to methods in the same class, subclasses, and classes in the same package as the member's class. A protected member's access is somewhere between private and public.

Let's look at a class with a protected member. Code Listing 10-18 shows the `GradedActivity2` class, which is a modification of the `GradedActivity` class presented earlier. In this class, the `score` field has been made protected instead of private.

**Code Listing 10-18** (`GradedActivity2.java`)

```
/**
   A class that holds a grade for a graded activity.
*/

public class GradedActivity2
{
   protected double score;  // Numeric score

   /**
      The setScore method sets the score field.
      @param s The value to store in score.
   */

   public void setScore(double s)
   {
      score = s;
   }

   /**
      The getScore method returns the score.
      @return The value stored in the score field.
   */

   public double getScore()
   {
      return score;
27    }
28
29    /**
30       The getGrade method returns a letter grade
31       determined from the score field.
32       @return The letter grade.
33    */
34
35    public char getGrade()
36    {
37       char letterGrade;
38
39       if (score >= 90)
40          letterGrade = 'A';
41       else if (score >= 80)
42          letterGrade = 'B';
43       else if (score >= 70)
44          letterGrade = 'C';
45       else if (score >= 60)
46          letterGrade = 'D';
47       else
48          letterGrade = 'F';
49
50       return letterGrade;
51    }
52 }
```

Because in line 7 the `score` field is declared as `protected`, any class that inherits from this class has direct access to it. The `FinalExam2` class, shown in Code Listing 10-19, is an example. This class is a modification of the `FinalExam` class, which was presented earlier. This class has a new method, `adjustScore`, which directly accesses the superclass's `score` field. If the contents of `score` have a fractional part of .5 or greater, the method rounds up `score` to the next whole number. The `adjustScore` method is called from the constructor.

**Code Listing 10-19** (`FinalExam2.java`)

```
1 /**
2    This class determines the grade for a final exam.
3    The numeric score is rounded up to the next whole
4    number if its fractional part is .5 or greater.
5 */
6
7 public class FinalExam2 extends GradedActivity2
8 {
9    private int numQuestions;    // Number of questions
```

```
   private double pointsEach;   // Points for each question
   private int numMissed;       // Number of questions missed

   /**
      The constructor sets the number of questions on the
      exam and the number of questions missed.
      @param questions The number of questions.
      @param missed The number of questions missed.
   */

   public FinalExam2(int questions, int missed)
   {
      double numericScore;        // To hold a numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the inherited setScore method to
      // set the numeric score.
      setScore(numericScore);

      // Adjust the score.
      adjustScore();
   }

   /**
      The getPointsEach method returns the number of
      points each question is worth.
      @return The value in the pointsEach field.
   */

   public double getPointsEach()
   {
      return pointsEach;
   }

   /**
      The getNumMissed method returns the number of
      questions missed.
      @return The value in the numMissed field.
   */

   public int getNumMissed()
   {
      return numMissed;
   }

   /**
      The adjustScore method adjusts a numeric score.
      If score is within 0.5 points of the next whole
      number, it rounds the score up.
   */

   private void adjustScore()
   {
      double fraction;

      // Get the fractional part of the score.
      fraction = score - (int) score;

      // If the fractional part is .5 or greater,
      // round the score up to the next whole number.
      if (fraction >= 0.5)
         score = score + (1.0 - fraction);
   }
}
```

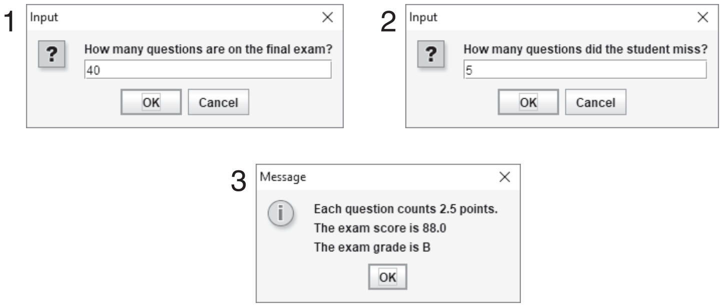
The program in Code Listing 10-20 demonstrates the class. Figure 10-12 shows an example of interaction with the program.

**Code Listing 10-20** **(ProtectedDemo.java)**

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the FinalExam2 class,
   which extends the GradedActivity2 class.
*/

public class ProtectedDemo
{
   public static void main(String[] args)
   {
      String input;        // To hold input
      int questions;       // Number of questions
      int missed;          // Number of questions missed

      // Get the number of questions on the exam.
      input = JOptionPane.showInputDialog("How many " +
                     "questions are on the final exam?");
      questions = Integer.parseInt(input);

      // Get the number of questions the student missed.
      input = JOptionPane.showInputDialog("How many " +
                     "questions did the student miss?");
      missed = Integer.parseInt(input);

      // Create a FinalExam object.
      FinalExam2 exam = new FinalExam2(questions, missed);

      // Display the test results.
      JOptionPane.showMessageDialog(null,
           "Each question counts " + exam.getPointsEach() +
           " points.\nThe exam score is " +
           exam.getScore() + "\nThe exam grade is " +
           exam.getGrade());

      System.exit(0);
   }
}
```

In the example running of the program in Figure 10-12, the student missed 5 out of 40 questions. The unadjusted numeric score would be 87.5, but the `adjustScore` method rounded up the `score` field to 88.

**Figure 10-12** Interaction with the `ProtectedDemo.java` program (Oracle Corporate Counsel)



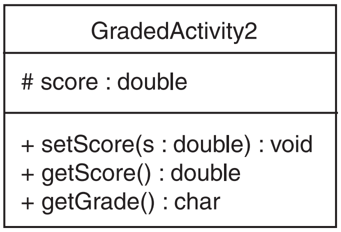
1 How many questions are on the final exam?
40

2 How many questions did the student miss?
5

3 Each question counts 2.5 points.
The exam score is 88.0
The exam grade is B

Protected class members may be denoted in a UML diagram with the # symbol. Figure 10-13 shows a UML diagram for the `GradedActivity2` class, with the `score` field denoted as protected.

**Figure 10-13** UML diagram for the `GradedActivity2` class



Although making a class member protected instead of private might make some tasks easier, you should avoid this practice when possible because any class that inherits from the class, or is in the same package, has unrestricted access to the protected member. It is always better to make all fields private and then provide public methods for accessing those fields.

## Package Access

If you do not provide an access specifier for a class member, the class member is given package access by default. This means that any method in the same package may access the member. Here is an example:

```
public class Circle
{
   double radius;
   int centerX, centerY;

   (Method definitions follow . . .)
}
```

In this class, the `radius`, `centerX`, and `centerY` fields were not given an access specifier, so the compiler grants them package access. Any method in the same package as the `Circle` class may directly access these members.

There is a subtle difference between protected access and package access. Protected members may be accessed by methods in the same package or in a subclass. This is true even if the subclass is in a different package. Members with package access, however, cannot be accessed by subclasses that are in a different package.

It is more likely that you will give package access to class members by accident than by design, because it is easy to forget the access specifier. Although there are circumstances under which package access can be helpful, you should normally avoid it. Be careful always to specify an access specifier for class members.

Tables 10-3 and 10-4 summarize how each of the access specifiers affects a class member's accessibility within and outside of the class's package.

**Table 10-3** Accessibility from within the class's package

| Access Specifier | Accessible to a subclass inside the same package? | Accessible to all other classes in the same package? |
|---|---|---|
| default (no modifier) | Yes | Yes |
| `public` | Yes | Yes |
| `protected` | Yes | Yes |
| `private` | No | No |

**Table 10-4** Accessibility from outside the class's package

| Access Specifier | Accessible to a subclass outside the same package? | Accessible to all other classes outside the same package? |
|---|---|---|
| default (no modifier) | No | No |
| `public` | Yes | Yes |
| `protected` | Yes | No |
| `private` | No | No |

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

10.11 When a class member is declared as `protected`, what code may access it?

10.12 What is the difference between private members and protected members?

10.13 Why should you avoid making class members protected when possible?

10.14 What is the difference between private access and package access?

10.15 Why is it easy to give package access to a class member by accident?

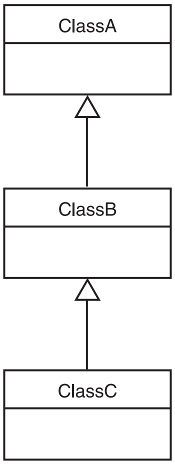
## 10.5 Chains of Inheritance

**CONCEPT: A superclass can also inherit from another class.**

Sometimes it is desirable to establish a chain of inheritance in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by Figure 10-14. In some cases, this chaining of classes goes on for many layers.

In Figure 10-14, `ClassC` inherits `ClassB`'s members, including the ones that `ClassB` inherited from `ClassA`. Let's look at an example of such a chain of inheritance. Consider the `PassFailActivity` class, shown in Code Listing 10-21, which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of 'P' for passing, or 'F' for failing.

**Figure 10-14** A chain of inheritance



**Code Listing 10-21** (`PassFailActivity.java`)

```
/**
   This class holds a numeric score and determines
   whether the score is passing or failing.
*/

public class PassFailActivity extends GradedActivity
{
   private double minPassingScore; // Minimum passing score

   /**
      The constructor sets the minimum passing score.
      @param mps The minimum passing score.
   */

   public PassFailActivity(double mps)
   {
      minPassingScore = mps;
   }

   /**
      The getGrade method returns a letter grade
      determined from the score field. This
      method overrides the superclass method.
      @return The letter grade.
   */

   @Override
   public char getGrade()
   {
```

```
30       char letterGrade;
31
32       if (super.getScore() >= minPassingScore)
33          letterGrade = 'P';
34       else
35          letterGrade = 'F';
36
37       return letterGrade;
38    }
39 }
```

The `PassFailActivity` constructor, in lines 15 through 18, accepts a `double` argument, which is the minimum passing grade for the activity. This value is stored in the `minPassingScore` field. The `getGrade` method, in lines 28 through 38, overrides the superclass method of the same name. This method returns a grade of `'P'` if the numeric score is greater-than or equal-to `minPassingScore`. Otherwise, the method returns a grade of `'F'`.

Suppose we wish to extend this class with another more specialized class. For example, the `PassFailExam` class, shown in Code Listing 10-22, determines a passing or failing grade for an exam. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

**Code Listing 10-22** (`PassFailExam.java`)

```
/**
   This class determines a passing or failing grade for
   an exam.
*/

public class PassFailExam extends PassFailActivity
{
   private int numQuestions;      // Number of questions
   private double pointsEach;     // Points for each question
   private int numMissed;         // Number of questions missed

   /**
      The constructor sets the number of questions, the
      number of questions missed, and the minimum passing
      score.
      @param questions The number of questions.
      @param missed The number of questions missed.
      @param minPassing The minimum passing score.
   */

   public PassFailExam(int questions, int missed,
                       double minPassing)
   {
      // Call the superclass constructor.
      super(minPassing);

      // Declare a local variable for the score.
      double numericScore;

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the superclass's setScore method to
      // set the numeric score.
      setScore(numericScore);
   }

   /**
      The getPointsEach method returns the number of
      points each question is worth.
      @return The value in the pointsEach field.
   */

   public double getPointsEach()
   {
      return pointsEach;
   }

   /**
      The getNumMissed method returns the number of
      questions missed.
      @return The value in the numMissed field.
   */

   public int getNumMissed()
   {
      return numMissed;
   }
}
```

The `PassFailExam` class inherits the `PassFailActivity` class's members, including the ones that `PassFailActivity` inherited from `GradedActivity`. The program in Code Listing 10-23 demonstrates the class.

**Code Listing 10-23** (`PassFailExamDemo.java`)

```
import java.util.Scanner;

/**
   This program demonstrates the PassFailExam class.
*/

public class PassFailExamDemo
{
   public static void main(String[] args)
   {
      int questions;       // Number of questions
      int missed;          // Number of questions missed
      double minPassing;   // Minimum passing score

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of questions on the exam.
      System.out.print("How many questions are " +
                       "on the exam? ");
      questions = keyboard.nextInt();

      // Get the number of questions missed.
      System.out.print("How many questions did " +
                       "the student miss? ");
      missed = keyboard.nextInt();

      // Get the minimum passing score.
      System.out.print("What is the minimum " +
                       "passing score? ");
      minPassing = keyboard.nextDouble();

      // Create a PassFailExam object.
      PassFailExam exam =
           new PassFailExam(questions, missed, minPassing);

      // Display the points for each question.
      System.out.println("Each question counts " +
                         exam.getPointsEach() + " points.");

      // Display the exam score.
      System.out.println("The exam score is " +
                         exam.getScore());

      // Display the exam grade.
```

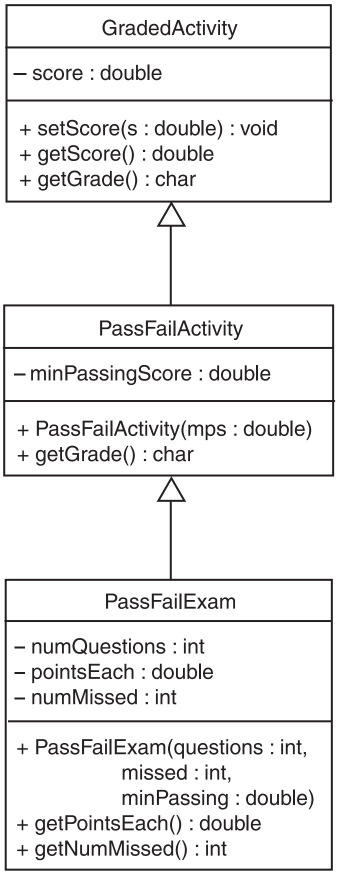
```
    System.out.println("The exam grade is " +
                       exam.getGrade());
  }
}
```

**Program Output with Example Input Shown in Bold**

```
How many questions are on the exam? 100 [Enter]
How many questions did the student miss? 25 [Enter]
What is the minimum passing score? 60 [Enter]
Each question counts 1.0 points.
The exam score is 75.0
The exam grade is P
```

Figure 10-15 is a UML diagram showing the inheritance relationship among the `GradedActivity`, `PassFailActivity`, and `PassFailExam` classes.

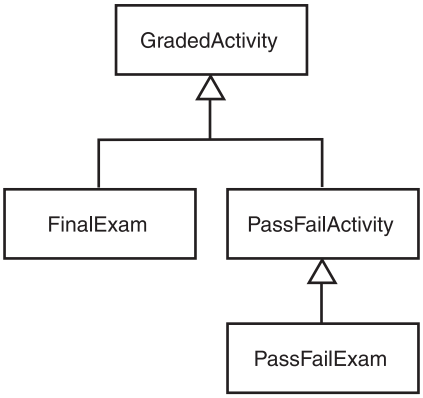
**Figure 10-15** The `GradedActivity`, `PassFailActivity`, and `PassFailExam` classes


GradedActivity
– score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
PassFailActivity
– minPassingScore : double
+ PassFailActivity(mps : double)
+ getGrade() : char
PassFailExam
– numQuestions : int
– pointsEach : double
– numMissed : int
+ PassFailExam(questions : int,
missed : int,
minPassing : double)
+ getPointsEach() : double
+ getNumMissed() : int

## Class Hierarchies

Classes often are depicted graphically in a class hierarchy. Like a family tree, a class hierarchy shows the inheritance relationships between classes. Figure 10-16 shows a class hierarchy for the `GradedActivity`, `FinalExam`, `PassFailActivity`, and `PassFailExam` classes. The more general classes are toward the top of the tree and the more specialized classes are toward the bottom.

**Figure 10-16** Class hierarchy



## 10.6 The `Object` Class

**CONCEPT: The Java API has a class named `Object`, which all other classes directly or indirectly inherit from.**

Every class in Java, including the ones in the API and the classes that you create, directly or indirectly inherits from a class named `Object`, which is part of the `java.lang` package. Here's how it happens: When a class does not use the `extends` key word to inherit from another class, Java automatically extends it from the `Object` class. For example, look at the following class declaration:

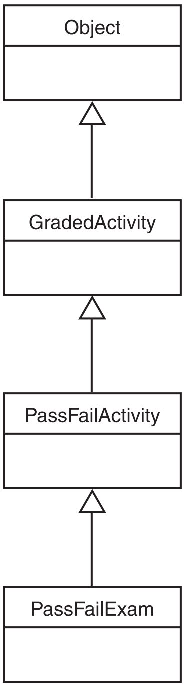
```
public class MyClass
{
    (Member Declarations . . .)
}
```

This class does not explicitly extend any other class, so Java treats it as though it were written as follows:

```
public class MyClass extends Object
{
    (Member Declarations . . .)
}
```

Ultimately, every class extends the `Object` class. Figure 10-17 shows how the `PassFailExam` class inherits from `Object`.

**Figure 10-17** The line of inheritance from `Object` to `PassFailExam`



Because every class directly or indirectly extends the `Object` class, every class inherits the `Object` class's members. Two of the most useful are the `toString` and `equals` methods. In Chapter 8 you learned that every class has a `toString` and an `equals` method, and now you know why! It is because those methods are inherited from the `Object` class.

In the `Object` class, the `toString` method returns a reference to a `String` containing the object's class name, followed by the `@` sign, followed by the object's hash code, which is a hexadecimal number. The `equals` method accepts a reference to an object as its argument. It returns `true` if the argument references the calling object. This is demonstrated in Code Listing 10-24.

**Code Listing 10-24** (`ObjectMethods.java`)

```
/**
   This program demonstrates the toString and equals
   methods that are inherited from the Object class.
*/

public class ObjectMethods
{
   public static void main(String[] args)
   {
      // Create two objects.
      PassFailExam exam1 =
                    new PassFailExam(0, 0, 0);
      PassFailExam exam2 =
                    new PassFailExam(0, 0, 0);

      // Send the objects to println, which
      // will call the toString method.
      System.out.println(exam1);
      System.out.println(exam2);

      // Test the equals method.
      if (exam1.equals(exam2))
         System.out.println("They are the same.");
      else
         System.out.println("They are not the same.");
   }
}
```

**Program Output**

```
PassFailExam@16f0472
PassFailExam@18d107f
They are not the same.
```

If you wish to change the behavior of either of these methods for a given class, you must override them in the class.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

10.16 Look at the following class definition:

```
public class ClassD extends ClassB
{
   (Member Declarations . . .)
}
```

Because `ClassD` inherits from `ClassB`, is it true that `ClassD` does not inherit from the `Object` class? Why or why not?

10.17 When you create a class, it automatically has a `toString` method and an equals method. Why?

## 10.7 Polymorphism

**CONCEPT: A superclass reference variable can reference objects of a subclass.**

VideoNote
Polymorphism

Look at the following statement that declares a reference variable named `exam`:

```
GradedActivity exam;
```

This statement tells us that the `exam` variable's data type is `GradedActivity`. Therefore, we can use the `exam` variable to reference a `GradedActivity` object, as shown in the following statement:

```
exam = new GradedActivity();
```

The `GradedActivity` class is also used as the superclass for the `FinalExam` class. Because of the "is-a" relationship between a superclass and a subclass, an object of the `FinalExam` class is not just a `FinalExam` object. It is also a `GradedActivity` object. (A final exam is a graded activity.) Because of this relationship, we can use a `GradedActivity` variable to reference a `FinalExam` object. For example, look at the following statement:

```
GradedActivity exam = new FinalExam(50, 7);
```

This statement declares `exam` as a `GradedActivity` variable. It creates a `FinalExam` object and stores the object's address in the `exam` variable. This statement is perfectly legal and will not cause an error message because a `FinalExam` object is also a `GradedActivity` object.

This is an example of polymorphism. The term *polymorphism* means the ability to take many forms. In Java, a reference variable is polymorphic because it can reference objects of types different from its own, as long as those types are subclasses of its type. All of the following declarations are legal because the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes inherit from `GradedActivity`:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

Although a `GradedActivity` variable can reference objects of any class that extends `GradedActivity`, there is a limit to what the variable can do with those objects. Recall that the `GradedActivity` class has three methods: `setScore`, `getScore`, and `getGrade`. So, a `GradedActivity` variable can be used to call only those three methods, regardless of the type of object the variable references. For example, look at the following code:

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore());      // This works.
System.out.println(exam.getGrade());      // This works.
System.out.println(exam.getPointsEach()); // ERROR! Won't work.
```

In this code, `exam` is declared as a `GradedActivity` variable and is assigned the address of a `PassFailExam` object. The `GradedActivity` class has only the `setScore`, `getScore`, and `getGrade` methods, so those are the only methods that the `exam` variable knows how to execute. The last statement in this code is a call to the `getPointsEach` method, which is defined in the `PassFailExam` class. Because the `exam` variable only knows about methods in the `GradedActivity` class, it cannot execute this method.

## Polymorphism and Dynamic Binding

When a superclass variable references a subclass object, a potential problem exists. What if the subclass has overridden a method in the superclass, and the variable makes a call to that method? Does the variable call the superclass's version of the method, or the subclass's version? For example, look at the following code:

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

Recall that the `PassFailActivity` class extends the `GradedActivity` class, and it overrides the `getGrade` method. When the last statement calls the `getGrade` method, does it call the `GradedActivity` class's version (which returns `'A'`, `'B'`, `'C'`, `'D'`, or `'F'`) or does it call the `PassFailActivity` class's version (which returns `'P'` or `'F'`)?

Recall from Chapter 6 that the process of matching a method call with the correct method definition is known as binding. Java performs dynamic binding or late binding when a variable contains a polymorphic reference. This means that the Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references. So, it is the object's type that determines which method is called, not the variable's type. In this case, the `exam` variable references a `PassFailActivity` object, so the `PassFailActivity` class's version of the `getGrade` method is called. The last statement in this code will display a grade of `P`.

The program in Code Listing 10-25 demonstrates polymorphic behavior. It declares an array of `GradedActivity` variables, and then assigns the addresses of objects of various types to the elements of the array.

**Code Listing 10-25** (`Polymorphic.java`)

```
/**
   This program demonstrates polymorphic behavior.
*/

public class Polymorphic
{
   public static void main(String[] args)
   {
      // Create an array of GradedActivity references.
      GradedActivity[] tests = new GradedActivity[3];

      // The first test is a regular exam with a
      // numeric score of 75.
      tests[0] = new GradedActivity();
      tests[0].setScore(95);

      // The second test is a pass/fail test. The
      // student missed 5 out of 20 questions, and
      // the minimum passing grade is 60.
      tests[1] = new PassFailExam(20, 5, 60);

      // The third test is the final exam. There were
      // 50 questions and the student missed 7.
      tests[2] = new FinalExam(50, 7);

      // Display the grades.
      for (int i = 0; i < tests.length; i++)
      {
         System.out.println("Test " + (i + 1) + ": " +
                           "score " + tests[i].getScore() +
                           ", grade " + tests[i].getGrade());
      }
   }
}
```

**Program Output**

```
Test 1: score 95.0, grade A
Test 2: score 75.0, grade P
Test 3: score 86.0, grade B
```

You can also use parameters to accept arguments to methods polymorphically. For example, look at the following method:

```
public static void displayGrades(GradedActivity g)
{
   System.out.println("Score " + g.getScore() +
                      ", grade " + g.getGrade());
}
```

This method's parameter, `g`, is a `GradedActivity` variable. But, it can be used to accept arguments of any type that inherit from `GradedActivity`. For example, the following code passes objects of the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes to the method:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
displayGrades(exam1);   // Pass a FinalExam object.
displayGrades(exam2);   // Pass a PassFailActivity object.
displayGrades(exam3);   // Pass a PassFailExam object.
```

## The "Is-a" Relationship Does Not Work in Reverse

It is important to note that the "is-a" relationship does not work in reverse. Although the statement "a final exam is a graded activity" is true, the statement "a graded activity is a final exam" is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = activity;    // ERROR!
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` variable. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the Java compiler will let you make such an assignment if you use a type cast, as shown here:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = (FinalExam) activity;  // Will compile but not run.
```

But, the program will crash when the assignment statement executes.

## The `instanceof` Operator

There is an operator in Java named `instanceof` that you can use to determine whether an object is an instance of a particular class. Here is the general form of an expression that uses the `instanceof` operator:

```
refVar instanceof ClassName
```

In the general form, *`refVar`* is a reference variable and *`ClassName`* is the name of a class. This is the form of a `boolean` expression that will return `true` if the object referenced by *`refVar`* is an instance of *`ClassName`*. Otherwise, the expression returns `false`. For example, the `if` statement in the following code determines whether the reference variable `activity` references a `GradedActivity` object:

```
GradedActivity activity = new GradedActivity();
if (activity instanceof GradedActivity)
   System.out.println("Yes, activity is a GradedActivity.");
else
   System.out.println("No, activity is not a GradedActivity.");
```

This code will display `"Yes, activity is a GradedActivity."`

The `instanceof` operator understands the "is-a" relationship that exists when a class inherits from another class. For example, look at the following code:

```
FinalExam exam = new FinalExam(20, 2);
if (exam instanceof GradedActivity)
   System.out.println("Yes, exam is a GradedActivity.");
else
   System.out.println("No, exam is not a GradedActivity.");
```

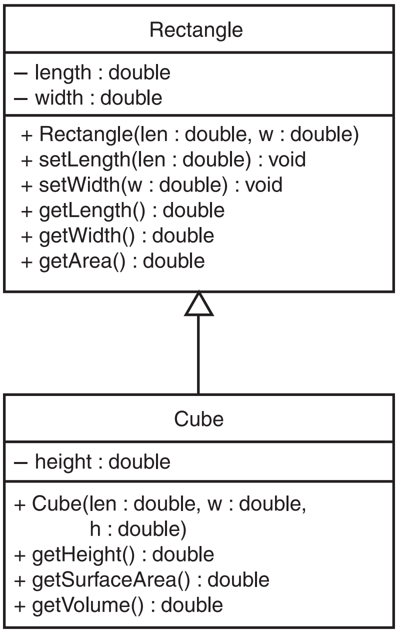
Even though the object referenced by `exam` is a `FinalExam` object, this code will display `"Yes, exam is a GradedActivity."` The `instanceof` operator returns `true` because `FinalExam` is a subclass of `GradedActivity`.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

10.18 Recall the `Rectangle` and `Cube` classes discussed earlier, as shown in Figure 10-18.

**Figure 10-18** `Rectangle` and Cube classes


Rectangle
– length : double
– width : double
+ Rectangle(len : double, w : double)
+ setLength(len : double) : void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double
Cube
– height : double
+ Cube(len : double, w : double, h : double)
+ getHeight() : double
+ getSurfaceArea() : double
+ getVolume() : double

a) Is the following statement legal or illegal? If it is illegal, why?

```
Rectangle r = new Cube(10, 12, 5);
```

b) If you determined that the statement in part a is legal, are the following statements legal or illegal? (Indicate legal or illegal for each statement.)

```
System.out.println(r.getLength());
System.out.println(r.getWidth());
System.out.println(r.getHeight());
System.out.println(r.getSurfaceArea());
```

c) Is the following statement legal or illegal? If it is illegal, why?

```
Cube c = new Rectangle(10, 12);
```

## 10.8 Abstract Classes and Abstract Methods

**CONCEPT: An abstract class is not instantiated, but other classes extend it. An abstract method has no body and must be overridden in a subclass.**

An abstract method is a method that appears in a superclass, but expects to be overridden in a subclass. An abstract method has only a header and no body. Here is the general format of an abstract method header:

```
AccessSpecifier abstract ReturnType MethodName(ParameterList);
```

Notice that the key word `abstract` appears in the header, and that the header ends with a semicolon. There is no body for the method. Here is an example of an abstract method header:

```
public abstract void setValue(int value);
```

When an abstract method appears in a class, the method must be overridden in a subclass. If a subclass fails to override the method, an error will result. Abstract methods are used to ensure that a subclass implements the method.

When a class contains an abstract method, you cannot create an instance of the class. Abstract methods are commonly used in abstract classes. An abstract class is not instantiated itself, but serves as a superclass for other classes. The abstract class represents the generic or abstract form of all the classes that inherit from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of airplanes: two different models of prop-driven planes and one commuter jet model. The computer software that catalogs the planes might use an abstract class named `Airplane`. That class has members representing the common characteristics of all airplanes. In addition, the software has classes for each of the three specific airplane models the factory manufactures. These classes all extend the `Airplane` class, and they have members representing the unique characteristics of each type of plane. The `Airplane` class is never instantiated, but is used as a superclass for the other classes.

A class becomes abstract when you place the `abstract` key word in the class definition. Here is the general format:

```
AccessSpecifier abstract class ClassName
```

An abstract class is not instantiated, but other classes extend it. An abstract method has no body and must be overridden in a subclass.

For example, look at the following abstract class `Student` shown in Code Listing 10-26. It holds data common to all students, but does not hold all the data needed for students of specific majors.

**Code Listing 10-26** (`Student.java`)

```
/**
   The Student class is an abstract class that holds
   general data about a student. Classes representing
   specific types of students should inherit from
   this class.
*/

public abstract class Student
{
   private String name;        // Student name
   private String idNumber;    // Student ID
   private int yearAdmitted;   // Year admitted

   /**
      The constructor sets the student's name,
      ID number, and year admitted.
      @param n The student's name.
      @param id The student's ID number.
      @param year The year the student was admitted.
   */

   public Student(String n, String id, int year)
   {
      name = n;
      idNumber = id;
      yearAdmitted = year;
   }

   /**
      The toString method returns a String containing
      the student's data.
      @return A reference to a String.
   */

   public String toString()
   {
      String str;

      str = "Name: " + name
         + "\nID Number: " + idNumber
         + "\nYear Admitted: " + yearAdmitted;
      return str;
   }

   /**
      The getRemainingHours method is abstract.
      It must be overridden in a subclass.
      @return The hours remaining for the student.
   */

   public abstract int getRemainingHours();
}
```

The `Student` class contains fields for storing a student's name, ID number, and year admitted. It also has a constructor, a `toString` method, and an abstract method named `getRemainingHours`.

This abstract method must be overridden in classes that inherit from the `Student` class. The idea behind this method is that it returns the number of hours remaining for a student to take in his or her major. It was made abstract because this class is intended to be the base for other classes that represent students of specific majors. For example, a `CompSciStudent` class might hold the data for a computer science student, and a `BiologyStudent` class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the `CompSciStudent` class will calculate the number of hours remaining to be taken differently than the `BiologyStudent` class. Let's look at an example of the `CompSciStudent` class, which is shown in Code Listing 10-27.

**Code Listing 10-27** (`CompSciStudent.java`)

```
/**
   This class holds data for a computer science student.
*/

public class CompSciStudent extends Student
{
   // Required hours
   private final int MATH_HOURS = 20;   // Math hours
   private final int CS_HOURS = 40;     // Comp sci hours
   private final int GEN_ED_HOURS = 60; // Gen ed hours

   // Hours taken
   private int mathHours;  // Math hours taken
   private int csHours;    // Comp sci hours taken
   private int genEdHours; // General ed hours taken

   /**
      The Constructor sets the student's name,
      ID number, and the year admitted.
      @param n The student's name.
      @param id The student's ID number.
      @param year The year the student was admitted.
   */

   public CompSciStudent(String n, String id, int year)
   {
      super(n, id, year);
   }

   /**
      The setMathHours method sets the number of
      math hours taken.
      @param math The math hours taken.
   */

   public void setMathHours(int math)
   {
      mathHours = math;
   }

   /**
```

```
      The setCsHours method sets the number of
      computer science hours taken.
      @param cs The computer science hours taken.
   */

   public void setCsHours(int cs)
   {
      csHours = cs;
   }

   /**
      The setGenEdHours method sets the number of
      general ed hours taken.
      @param genEd The general ed hours taken.
   */

   public void setGenEdHours(int genEd)
   {
      genEdHours = genEd;
   }

   /**
      The getRemainingHours method returns the
      the number of hours remaining to be taken.
      @return The hours remaining for the student.
   */

   @Override
   public int getRemainingHours()
   {
      int reqHours,        // Total required hours
          remainingHours;  // Remaining hours

      // Calculate the required hours.
      reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;

      // Calculate the remaining hours.
      remainingHours = reqHours - (mathHours + csHours
                         + genEdHours);

      return remainingHours;
   }

   /**
      The toString method returns a string containing
      the student's data.
      @return A reference to a String.
   */
```

```
 90
 91    @Override
 92    public String toString()
 93    {
 94       String str;
 95
 96       str = super.toString() +
 97          "\nMajor: Computer Science" +
 98          "\nMath Hours Taken: " + mathHours +
 99          "\nComputer Science Hours Taken: " + csHours +
100          "\nGeneral Ed Hours Taken: " + genEdHours;
101
102       return str;
103    }
104 }
```

The `CompSciStudent` class, which extends the `Student` class, declares the following `final` integer fields in lines 8 through 10: `MATH_HOURS`, `CS_HOURS`, and `GEN_ED_HOURS`. These fields hold the required number of math, computer science, and general education hours for a computer science student. It also declares the following fields in lines 13 through 15: `mathHours`, `csHours`, and `genEdHours`. These fields hold the number of math, computer science, and general education hours taken by the student. Mutator methods are provided to store values in these fields. In addition, the class overrides the `toString` method and the abstract `getRemainingHours` method. The program in Code Listing 10-28 demonstrates the class.

**Code Listing 10-28** (`CompSciStudentDemo.java`)

```
/**
   This program demonstrates the CompSciStudent class.
*/

public class CompSciStudentDemo
{
   public static void main(String[] args)
   {
      // Create a CompSciStudent object.
      CompSciStudent csStudent =
              new CompSciStudent("Jennifer Haynes",
                                 "167W98337", 2015);

      // Store values for math, CS, and gen ed hours.
      csStudent.setMathHours(12);
      csStudent.setCsHours(20);
      csStudent.setGenEdHours(40);

      // Display the student's data.
      System.out.println(csStudent);

      // Display the number of remaining hours.
      System.out.println("Hours remaining: " +
                         csStudent.getRemainingHours());
   }
}
```

**Program Output**

```
Name: Jennifer Haynes
ID Number: 167W98337
Year Admitted: 2015
Major: Computer Science
Math Hours Taken: 12
Computer Science Hours Taken: 20
General Ed Hours Taken: 40
Hours remaining: 48
```

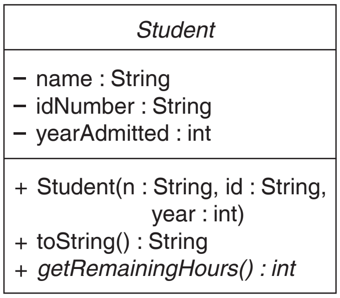
Remember the following points about abstract methods and classes:

- Abstract methods and abstract classes are defined with the `abstract` key word.
- Abstract methods have no body, and their header must end with a semicolon.
- An abstract method must be overridden in a subclass.
- When a class contains an abstract method, it cannot be instantiated. It must serve as a superclass.
- An abstract class cannot be instantiated. It must serve as a superclass.

## Abstract Classes in UML

Abstract classes are drawn like regular classes in UML, except the name of the class and the names of abstract methods are shown in italics. For example, Figure 10-19 shows a UML diagram for the `Student` class.

**Figure 10-19** UML diagram for the `Student` class


*Student*
− name : String
− idNumber : String
− yearAdmitted : int
+ Student(n : String, id : String, year : int)
+ toString() : String
+ *getRemainingHours() : int*

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

10.19 What is the purpose of an abstract method?

10.20 If a subclass extends a superclass with an abstract method, what must you do in the subclass?

10.21 What is the purpose of an abstract class?

10.22 If a class is defined as abstract, what can you not do with the class?

## 10.9 Interfaces

**CONCEPT:** **An interface specifies behavior for a class.**

In its simplest form, an interface is like a class that contains only abstract methods. An interface cannot be instantiated. Instead, it is *implemented* by other classes. When a class implements an interface, the class must override the methods that are specified by the interface.

An interface looks similar to a class, except the key word `interface` is used instead of the key word `class`, and the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons. Here is the general format of an interface definition:

```
public interface InterfaceName
{
  (Method headers . . .)
}
```

Code Listing 10-29 shows an example of an interface named `Displayable`. In line 3, the interface specifies a `void` method named `display()`.

**Code Listing 10-29** `(Displayable.java)`

```
1  public interface Displayable
2  {
3     void display();
4  }
```

Notice that the `display` method header in line 3 does not have an access specifier. This is because all methods in an interface are implicitly `public`. You can optionally write `public` in the method header, but most programmers leave it out because all interface methods must be `public`.

Any class that implements the `Displayable` interface shown in Code Listing 10-29 *must* provide an implementation of the `display` method (with the exact signatures specified by the interface, and with the same return type). The `Person` class shown in Code Listing 10-30 is an example.

### Code Listing 10-30 (`Person.java`)

```java
 1 public class Person implements Displayable
 2 {
 3    private String name;
 4
 5    // Constructor
 6    public Person(String n)
 7    {
 8       name = n;
 9    }
10
11    // display method
12    public void display()
13    {
14       System.out.println("My name is " + name);
15    }
16 }
```

When you want a class to implement an interface, you use the `implements` key word in the class header. Notice in line 1 of Code Listing 10-30, the `Person` class header ends with the clause `implements Displayable`. Because the `Person` class implements the `Displayable` interface, it must provide an implementation of the interface's `display` method. This is done in lines 12 through 15 of the `Person` class. The program in Code Listing 10-31 demonstrates the `Person` class.

### Code Listing 10-31 (`InterfaceDemo.java`)

```java
/**
   This program demonstrates a class that implements
   the Displayable interface.
*/

public class InterfaceDemo
{
   public static void main(String[] args)
   {
      // Create an instance of the Person class.
      Person p = new Person("Antonio");

      // Call the object's display method.
      p.display();
   }
}
```

**Program Output**

```
My name is Antonio
```

## An Interface is a Contract

When a class implements an interface, it is agreeing to provide all of the methods that are specified by the interface. It is often said that an interface is like a “contract,” and when a class implements an interface it must adhere to the contract.

For example, Code Listing 10-32 shows an interface named `Relatable`, which is intended to be used with the `GradedActivity` class presented earlier. This interface has three method headers: `equals`, `isGreater`, and `isLess`. Notice that each method accepts a `GradedActivity` object as its argument.

**Code Listing 10-32** (`Relatable.java`)

```
/**
   Relatable interface
*/

public interface Relatable
{
   boolean equals(GradedActivity g);
   boolean isGreater(GradedActivity g);
   boolean isLess(GradedActivity g);
}
```

You might have guessed that the `Relatable` interface is named “Relatable” because it specifies methods that presumably, make relational comparisons with `GradedActivity` objects. The intent is to make any class that implements this interface “relatable” with `GradedActivity` objects by ensuring that it has an `equals`, an `isGreater`, and an `isLess` method that perform relational comparisons. But, the interface only specifies the headers for these methods, not what the methods should do. Although the programmer of a class that implements the `Relatable` interface can choose what those methods should do, he or she should provide methods that comply with this intent.

Code Listing 10-33 shows the code for the `FinalExam3` class, which implements the `Relatable` interface. The `equals`, `isGreater`, and `isLess` methods compare the calling object with the object passed as an argument. The program in Code Listing 10-34 demonstrates the class.

**Code Listing 10-33** (`FinalExam3.java`)

```
/**
   This class determines the grade for a final exam.
*/

public class FinalExam3 extends GradedActivity implements Relatable
{
   private int numQuestions;  // Number of questions
   private double pointsEach; // Points for each question
   private int numMissed;     // Questions missed

   /**
      The constructor sets the number of questions on the
      exam and the number of questions missed.
      @param questions The number of questions.
      @param missed The number of questions missed.
   */

   public FinalExam3(int questions, int missed)
   {
      double numericScore;  // To hold a numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the inherited setScore method to
      // set the numeric score.
      setScore(numericScore);
   }

   /**
      The getPointsEach method returns the number of
      points each question is worth.
      @return The value in the pointsEach field.
   */

   public double getPointsEach()
   {
      return pointsEach;
   }

   /**
      The getNumMissed method returns the number of
      questions missed.
      @return The value in the numMissed field.
   */

   public int getNumMissed()
   {
      return numMissed;
   }

```

```
/**
   The equals method compares the calling object
   to the argument object for equality.
   @return true if the calling
   object's score is equal to the argument's
   score.
*/

public boolean equals(GradedActivity g)
{
   boolean status;

   if (this.getScore() == g.getScore())
      status = true;
   else
      status = false;

   return status;
}

/**
   The isGreater method determines whether the calling
   object is greater than the argument object.
   @return true if the calling object's score is
   greater than the argument object's score.
*/

public boolean isGreater(GradedActivity g)
{
   boolean status;

   if (this.getScore() > g.getScore())
      status = true;
   else
      status = false;

   return status;
}

/**
   The isLess method determines whether the calling
   object is less than the argument object.
   @return true if the calling object's score is
   less than the argument object's score.
*/

public boolean isLess(GradedActivity g)
{
      boolean status;

      if (this.getScore() < g.getScore())
         status = true;
      else
         status = false;

      return status;
   }
}
```

**Code Listing 10-34** `(RelatableExams.java)`

```
/**
   This program demonstrates the FinalExam3 class which
   implements the Relatable interface.
*/

public class RelatableExams
{
   public static void main(String[] args)
   {
      // Exam #1 had 100 questions and the student
      // missed 20 questions.
      FinalExam3 exam1 = new FinalExam3(100, 20);

      // Exam #2 had 100 questions and the student
      // missed 30 questions.
      FinalExam3 exam2 = new FinalExam3(100, 30);

      // Display the exam scores.
      System.out.println("Exam 1: " + exam1.getScore());
      System.out.println("Exam 2: " + exam2.getScore());

      // Compare the exam scores.
      if (exam1.equals(exam2))
         System.out.println("The exam scores are equal.");

      if (exam1.isGreater(exam2))
         System.out.println("The Exam 1 score is the highest.");

      if (exam1.isLess(exam2))
         System.out.println("The Exam 1 score is the lowest.");
   }
}
```

**Program Output**

```
Exam 1: 80.0
Exam 2: 70.0
The Exam 1 score is the highest.
```

## Fields in Interfaces

An interface can contain field declarations, but all fields in an interface are treated as `final` and `static`. Because they automatically become `final`, you must provide an initialization value. For example, look at the following interface definition:

```
public interface Doable
{
   int FIELD1 = 1;
   int FIELD2 = 2;
   (Method headers . . .)
}
```

In this interface, `FIELD1` and `FIELD2` are `final static int` variables. Any class that implements this interface has access to these variables.

## Implementing Multiple Interfaces

You might be wondering why we need both abstract classes and interfaces, since they are so similar to each other. The reason is that a class can extend only one superclass, but Java allows a class to implement multiple interfaces. When a class implements multiple interfaces, it must provide the methods specified by all of them.

To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the `implements` key word. Here is the first line of an example of a class that implements multiple interfaces:

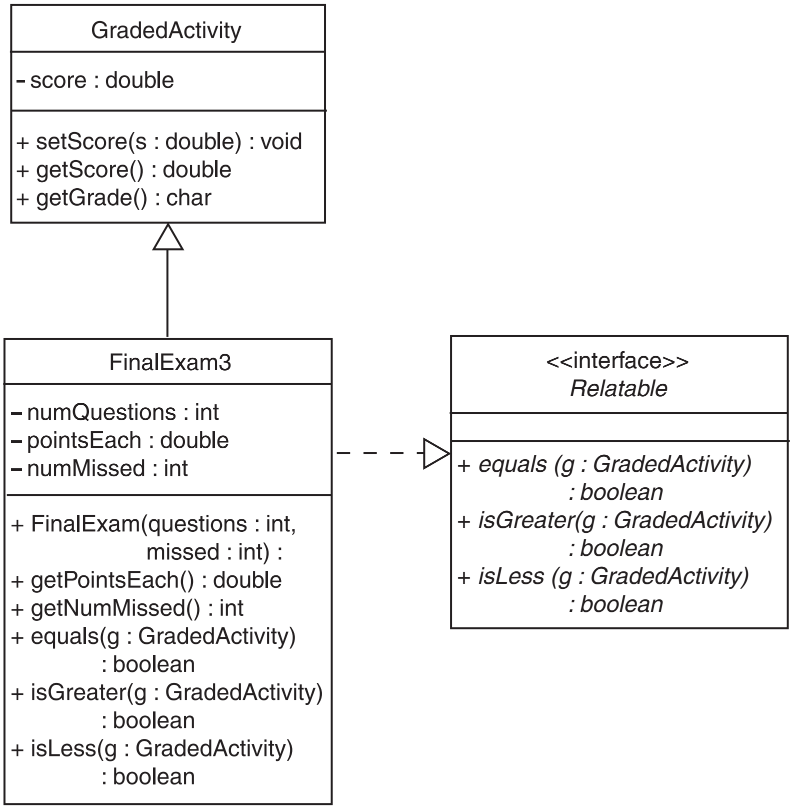
```
public class MyClass implements Interface1,
                                Interface2,
                                Interface3
```

This class implements three interfaces: `Interface1`, `Interface2`, and `Interface3`.

## Interfaces in UML

In a UML diagram, an interface is drawn like a class, except the interface name and the method names are italicized, and the <<interface>> tag is shown above the interface name. The relationship between a class and an interface is known as a realization relationship (the class realizes the interfaces). You show a realization relationship in a UML diagram by connecting a class and an interface with a dashed line that has an open arrowhead at one end. The arrowhead points to the interface. This depicts the realization relationship. Figure 10-20 is a UML diagram showing the relationships among the `GradedActivity` class, the `FinalExam3` class, and the `Relatable` interface.

**Figure 10-20** Realization relationship in a UML diagram


GradedActivity
- score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
FinalExam3
- numQuestions : int
- pointsEach : double
- numMissed : int
+ FinalExam(questions : int, missed : int) :
+ getPointsEach() : double
+ getNumMissed() : int
+ equals(g : GradedActivity) : boolean
+ isGreater(g : GradedActivity) : boolean
+ isLess(g : GradedActivity) : boolean
<<interface>>
Relatable
+ equals (g : GradedActivity) : boolean
+ isGreater(g : GradedActivity) : boolean
+ isLess (g : GradedActivity) : boolean

## Default Methods

Beginning in Java 8, interfaces may have *default methods*. A default method is an interface method that has a body. Code Listing 10-35 shows another version of the `Displayable` interface, in which the `display` method is a default method.

**Code Listing 10-35** (`Displayable.java`)

```
1 public interface Displayable
2 {
3    default void display()
4    {
5       System.out.println("This is the default display method.");
6    }
7 }
```

Notice in line 3 that the method header begins with the key word `default`. This is required for an interface method that has a body. When a class implements an interface with a default method, the class can override the default method, but it is not required to. For example, the

`Person` class shown in Code Listing 10-36 implements the `Displayable` interface, but does not override the `display` method. The program in Code Listing 10-37 instantiates the `Person` class, and calls the `display` method. As you can see from the program output, the code in the interface's default method is executed.

**Code Listing 10-36** (`Person.java`)

```
/**
   This class implements the Displayable
   interface, but does not override the
   default display method.
*/

public class Person implements Displayable
{
   private String name;

   // Constructor
   public Person(String n)
   {
      name = n;
   }
}
```

**Code Listing 10-37** (`InterfaceDemoDefaultMethod.java`)

```
/**
   This program demonstrates a class that implements
   the Displayable interface (with a default method).
*/

public class InterfaceDemoDefaultMethod
{
   public static void main(String[] args)
   {
      // Create an instance of the Person class.
      Person p = new Person("Antonio");

      // Call the object's display method.
      p.display();
   }
}
```

**Program Output**

```
This is the default display method.
```

**NOTE:** One of the benefits of having default methods is that they allow you to add new methods to an existing interface without causing errors in the classes that already implement the interface. Prior to Java 8, when you added a new method header to an existing interface, all of the classes that already implement that interface had to be rewritten to override the new method. Now you can add default methods to an interface, and if an existing class (that implements the interface) does not need the new method, you do not have to rewrite the class.

## Polymorphism and Interfaces

Just as you can create reference variables of a class type, Java allows you to create reference variables of an interface type. An interface reference variable can reference any object that implements that interface, regardless of its class type. This is another example of polymorphism. For example, look at the `RetailItem` interface in Code Listing 10-38.

**Code Listing 10-38** (`RetailItem.java`)

```java
/**
   RetailItem interface
*/

public interface RetailItem
{
   public double getRetailPrice();
}
```

This interface specifies only one method: `getRetailPrice`. Both the `CompactDisc` and `DvdMovie` classes, shown in Code Listings 10-39 and 10-40, implement this interface.

**Code Listing 10-39** (`CompactDisc.java`)

```java
/**
   Compact Disc class
*/

public class CompactDisc implements RetailItem
{
   private String title;         // The CD's title
   private String artist;        // The CD's artist
   private double retailPrice;   // The CD's retail price

   /**
      Constructor
      @param cdTitle The CD title.
      @param cdArtist The name of the artist.
      @param cdPrice The CD's price.
   */

   public CompactDisc(String cdTitle, String cdArtist,
                      double cdPrice)
   {
      title = cdTitle;
      artist = cdArtist;
      retailPrice = cdPrice;
   }

   /**
      getTitle method
      @return The CD's title.
   */

   public String getTitle()
   {
      return title;
   }

   /**
      getArtist method
      @return The name of the artist.
   */

   public String getArtist()
   {
      return artist;
   }

   /**
      getRetailPrice method (Required by the RetailItem
      interface)
      @return The retail price of the CD.
   */

   public double getRetailPrice()
   {
      return retailPrice;
   }
}
```

**Code Listing 10-40** (`DvdMovie.java`)

```
/**
   DvdMovie class
*/

```

```
public class DvdMovie implements RetailItem
{
   private String title;         // The DVD's title
   private int runningTime;      // Running time in minutes
   private double retailPrice;   // The DVD's retail price

   /**
      Constructor
      @param dvdTitle The DVD title.
      @param runTime The running time in minutes.
      @param dvdPrice The DVD's price.
   */

   public DvdMovie(String dvdTitle, int runTime,
                                double dvdPrice)
   {
      title = dvdTitle;
      runningTime = runTime;
      retailPrice = dvdPrice;
   }

   /**
      getTitle method
      @return The DVD's title.
   */

   public String getTitle()
   {
      return title;
   }

   /**
      getRunningTime method
      @return The running time in minutes.
   */

   public int getRunningTime()
   {
      return runningTime;
   }

   /**
      getRetailPrice method (Required by the RetailItem
      interface)
      @return The retail price of the DVD.
   */

   public double getRetailPrice()
   {
      return retailPrice;
   }
}
```

Because they implement the `RetailItem` interface, objects of these classes may be referenced by a `RetailItem` reference variable. The following code demonstrates:

```
RetailItem item1 = new CompactDisc("Songs From the Heart",
                                   "Billy Nelson",
                                   18.95);
RetailItem item2 = new DvdMovie("Planet X",
                                102,
                                22.95);
```

In this code, two `RetailItem` reference variables, `item1` and `item2`, are declared. The `item1` variable references a `CompactDisc` object and the `item2` variable references a `DvdMovie` object. This is possible because both the `CompactDisc` and `DvdMovie` classes implement the `RetailItem` interface. When a class implements an interface, an inheritance relationship known as interface inheritance is established. Because of this inheritance relationship, a `CompactDisc` object is a `RetailItem`, and likewise, a `DvdMovie` object is a `RetailItem`. Therefore, we can create `RetailItem` reference variables and have them reference `CompactDisc` and `DvdMovie` objects.

The program in Code Listing 10-41 demonstrates how an interface reference variable can be used as a method parameter.

**Code Listing 10-41** **(PolymorphicInterfaceDemo.java)**

```
/**
   This program demonstrates that an interface type may
   be used to create a polymorphic reference.
*/

public class PolymorphicInterfaceDemo
{
   public static void main(String[] args)
   {
      // Create a CompactDisc object.
      CompactDisc cd =
              new CompactDisc("Greatest Hits",
                              "Joe Looney Band",
                              18.95);
      // Create a DvdMovie object.
      DvdMovie movie =
              new DvdMovie("Wheels of Fury",
                           137, 12.95);

      // Display the CD's title.
      System.out.println("Item #1: " +
                         cd.getTitle());

      // Display the CD's price.
      showPrice(cd);

      // Display the DVD's title.
      System.out.println("Item #2: " +
                         movie.getTitle());

      // Display the DVD's price.
      showPrice(movie);
   }

   /**
      The showPrice method displays the price
      of a RetailItem object.
      @param item A reference to a RetailItem object.
   */

   private static void showPrice(RetailItem item)
   {
      System.out.printf("Price: $%,.2f\n", item.getRetailPrice());
   }
}
```

**Program Output**

```
Item #1: Greatest Hits
Price: $18.95
Item #2: Wheels of Fury
Price: $12.95
```

There are some limitations to using interface reference variables. As previously mentioned, you cannot create an instance of an interface. In addition, when an interface variable references an object, you can use the interface variable to call only the methods that are specified in the interface. For example, look at the following code:

```
// Reference a CompactDisc object with a RetailItem variable.
RetailItem item = new CompactDisc("Greatest Hits",
                                  "Joe Looney Band",
                                  18.95);

// Call the getRetailPrice method . . .
System.out.println(item.getRetailPrice()); // OK, this works.
// Attempt to call the getTitle method . . .
System.out.println(item.getTitle()); // ERROR! Will not compile!
```

The last line of code will not compile because the `RetailItem` interface specifies only one method: `getRetailPrice`. So, we cannot use a `RetailItem` reference variable to call any other method.

**TIP:** It is possible to cast an interface reference variable to the type of the object it references, and then call methods that are members of that type. The syntax is somewhat awkward, however. The statement that causes the compiler error in the example code could be rewritten as:

```
    System.out.println(((CompactDisc)item).getTitle());
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

10.23 What is the purpose of an interface?

10.24 How is an interface similar to an abstract class?

10.25 How is an interface different from an abstract class, or any class?

10.26 If an interface has fields, how are they treated?

10.27 Write the first line of a class named `Customer`, which implements an interface named `Relatable`.

10.28 Write the first line of a class named `Employee`, which implements interfaces named `Payable` and `Listable`.

## 10.10 Anonymous Inner Classes

**CONCEPT: An inner class is a class that is defined inside another class. An anonymous inner class is an inner class that has no name. An anonymous inner class must implement an interface, or extend another class.**

Sometimes you need a class that is simple, and to be instantiated only once in your code. When this is the case, you can use an *anonymous inner class*. An anonymous inner class is a class that has no name. It is called an inner class because it is defined inside another class. You use the `new` operator to simultaneously define an anonymous inner class and create an instance of it. Here is the general syntax for instantiating and defining an anonymous inner class:

```
new ClassOrInterfaceName() {

    (Fields and methods of the anonymous class...)
}
```

The `new` operator is followed by the name of an existing class or interface, followed by a set of parentheses. Next, you write the body of the class, enclosed in curly braces. The expression creates an object that is an instance of a class that either extends the specified superclass, or implements the specified interface. A reference to the object is returned. (Notice that you do *not* use the `extends` or `implements` key words in the expression.)

Before you look at an example, you must understand a few requirements and restrictions:

- An anonymous inner class must either implement an interface, or extend another class.
- If the anonymous inner class extends a superclass, the superclass's no-arg constructor is called when the object is created.
- An anonymous inner class must override all of the abstract methods specified by the interface it is implementing, or the superclass it is extending.
- Because an anonymous inner class's definition is written inside a method, it can access that method's local variables, but only if they are declared `final`, or if they are effectively `final`. (An effectively `final` variable is a variable whose value is never changed.) A compiler error will result if an anonymous inner class tries to use a variable that is not `final`, or not effectively `final`.

Let's look at an example of an anonymous inner class that implements an interface. Suppose we have the interface shown in Code Listing 10-42.

**Code Listing 10-42** `(IntCalculator.java)`

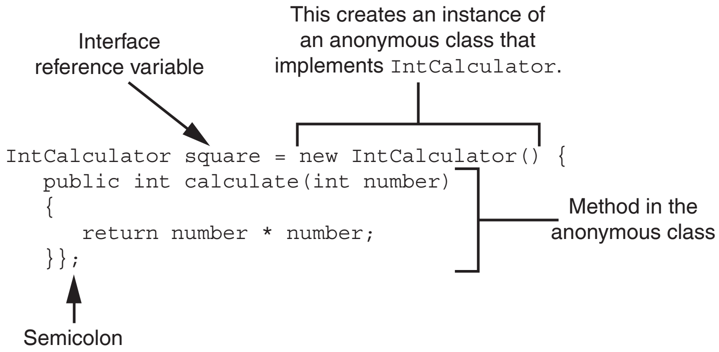
```
interface IntCalculator
{
   int calculate(int number);
}
```

The name of the interface is `IntCalculator`, and it specifies a method named `calculate`. The `calculate` method accepts an `int` argument, and returns an `int` value. Suppose we want to define a class that implements the `IntCalculator` interface, and overrides the `calculate` method so that it returns the square of the argument that is passed to it. The following code snippet shows how:

```
IntCalculator square = new IntCalculator() {
   public int calculate(int number)
   {
      return number * number;
   }};
```

The first line of the code snippet declares a variable named `square`, that is an `IntCalculator` reference variable (meaning that it can refer to any object that implements `IntCalculator`). On the right side of the = sign, the expression `new IntCalculator()` creates an instance of an anonymous class that implements the `IntCalculator` interface. The body of the anonymous class appears next, enclosed inside curly braces. In the class body, the `calculate` method is overridden. Because this is a complete statement, it ends with a semicolon. Figure 10-21 illustrates the different parts of the statement, and Code Listing 10-43 shows a complete program that uses it.

**Figure 10-21** Creating an instance of an anonymous inner class



```
IntCalculator square = new IntCalculator() {
   public int calculate(int number)
   {
      return number * number;
   }};
```

Interface reference variable
This creates an instance of an anonymous class that implements IntCalculator.
Method in the anonymous class
Semicolon

**Code Listing 10-43** (AnonymousClassDemo.java)

```
import java.util.Scanner;

/**
   This program demonstrates an anonymous inner class.
*/

public class AnonymousClassDemo
{
   public static void main(String[] args)
   {
      int num;

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create an object that implements IntCalculator.
      IntCalculator square = new IntCalculator() {
         public int calculate(int number)
         {
            return number * number;
         }};

      // Get a number from the user.
      System.out.print("Enter an integer number: ");
      num = keyboard.nextInt();
26
27       // Display the square of the number.
28       System.out.println("The square is " + square.calculate(num));
29    }
30 }
```

**Program Output with Example Input Shown in Bold**

```
Enter an integer number: 5 [Enter]
The square is 25
```

Let's take a closer look at the program:

- Line 11 declares an `int` variable named `num`, which will be used to hold user input.
- Line 14 creates a `Scanner` object for keyboard input.
- Lines 17 through 21 instantiate an anonymous inner class that implements the `IntCalculator` interface. A variable named `square` is used to reference the object. In the class body, lines 18 through 21 override the `calculate` method to return the square of the method's argument.
- Line 24 prompts the user to enter an integer number, and line 25 reads the number from the keyboard. The number is assigned to the `num` variable.
- Line 28 calls the `square` object's `calculate` method, and displays the return value in a message.

## 10.11 Functional Interfaces and Lambda Expressions

**CONCEPT: A functional interface is an interface that has one abstract method. You can use a special type of expression, known as a lambda expression, to create an object that implements a functional interface.**

Java 8 introduces two new features that work together to simplify code, particularly in situations where you might use anonymous inner classes. These new features are functional interfaces and lambda expressions. A *functional interface* is simply an interface that has one abstract method. For example, let's take another look at the `IntCalculator` interface that we previously discussed in the section on anonymous classes. Because it has only one abstract method, it is considered a functional interface. For your convenience, Code Listing 10-44 shows the code for the interface.

**Code Listing 10-44** (`IntCalculator.java`)

```
interface IntCalculator
{
   int calculate(int number);
}
```

The name of the interface is `IntCalculator`, and it specifies one method named `calculate`. The `calculate` method accepts an `int` argument, and returns an `int` value.

Because `IntCalculator` is a functional interface, we do not have to go to the trouble of defining a class that implements the interface. We do not even have to use an anonymous inner class. Instead, we can use a *lambda expression* to create an object that implements the interface, and overrides its abstract method.

You can think of a lambda expression as an anonymous method, or a method with no name. Like regular methods, lambda expressions can accept arguments and return values. Here is the general format of a simple lambda expression that accepts one argument, and returns a value:

```
parameter -> expression
```

In this general format, the lambda expression begins with a parameter variable, followed by the *lambda operator* (`->`), followed by an expression that has a value. Here is an example:

```
x -> x * x
```

The `x` that appears on the left side of the `->` operator is the name of a parameter variable, and the expression `x * x` that appears on the right side of the -> operator is the value that is returned.

This lambda expression works like a method that has a parameter variable named `x`, and it returns the value of `x * x`.

We can use this lambda expression to create an object that implements the `IntCalculator` interface. Here is an example:

```
IntCalculator square = x -> x * x;
```

On the left side of the = operator we declare an `IntCalculator` reference variable named `square`. On the right side of the = operator we have a lambda expression that creates an object with the following characteristics:

- Because we are assigning the object to an `IntCalculator` reference variable, the object automatically implements the `IntCalculator` interface.
- Because the `IntCalculator` interface has only one abstract method (named `calculate`), the lambda expression will be used to implement that one method.
- The parameter `x` that is used in the lambda expression represents the argument that is passed to the `calculate` method. We do not have to specify the data type of `x` because the compiler will determine it. Because the `calculate` method (in `theIntCalculator` interface) has an `int` parameter, the `x` parameter in the lambda expression will automatically become an `int`.
- The expression `x * x` is the value that is returned from the `calculate` method.

Code Listing 10-45 shows a complete program that uses the previously shown statement to create an object.

**Code Listing 10-45** (`LambdaDemo.java`)

```
import java.util.Scanner;

/**
```

```
 4   This program demonstrates a simple
 5   lambda expression.
 6 */
 7
 8 public class LambdaDemo
 9 {
10    public static void main(String[] args)
11    {
12       int num;
13
14       // Create a Scanner object for keyboard input.
15       Scanner keyboard = new Scanner(System.in);
16
17       // Create an object that implements IntCalculator.
18       IntCalculator square = x -> x * x;
19
20       // Get a number from the user.
21       System.out.print("Enter an integer number: ");
22       num = keyboard.nextInt();
23
24       // Display the square of the number.
25       System.out.println("The square is " + square.calculate(num));
26    }
27 }
```

**Program Output with Example Input Shown in Bold**

```
Enter an integer number: 5 [Enter]
The square is 25
```

Let's take a closer look at the program:

- Line 12 declares an `int` variable named `num`, which will be used to hold user input.
- Line 15 creates a `Scanner` object for keyboard input.
- Line 18 uses a lambda expression to create an object that implements the `IntCalculator` interface. A variable named `square` is used to reference the object. The object's `calculate` method will return the square of the method's argument.
- Line 21 prompts the user to enter an integer number, and line 22 reads the number from the keyboard. The number is assigned to the `num` variable.
- Line 25 calls the `square` object's `calculate` method, and displays the return value in a message.

Lambda expressions provide a way to easily create and instantiate anonymous inner classes. If you compare Code Listing 10-45 with the program shown in Code Listing 10-43, you can see that the lambda expression is much more concise than the anonymous inner class declaration.

### Lambda Expressions That Do Not Return a Value

If a functional interface's abstract method is `void` (does not return a value), any lambda expression that you use with the interface should also be `void`. Here is an example:

```
x -> System.out.println(x);
```

This lambda expression has a parameter, `x`. When the expression is invoked, it displays the value of `x`.

### Lambda Expressions with Multiple Parameters

If a functional interface's abstract method has multiple parameters, any lambda expression that you use with the interface must also have multiple parameters. To use more than one parameter in a lambda expression, simply write a comma-separated list and enclose the list in parentheses. Here is an example:

```
(a, b) -> a + b;
```

This lambda expression has two parameters, `a` and `b`. The expression returns the value of `a + b`.

### Lambda Expressions with No Parameters

If a functional interface's abstract method has no parameters, any lambda expression that you use with the interface must also have no parameters. Simply write a set of empty parentheses as the parameter list, as shown here:

```
() -> System.out.println();
```

When this lambda expression is invoked, it simply prints a blank line.

### Explicitly Declaring a Parameter's Data Type

You do not have to specify the data type of a lambda expression's parameter because the compiler will determine it from the interface's abstract method header. However, you can explicitly declare the data type of a parameter, if you wish. Here is an example:

```
(int x) -> x * x;
```

Note that the parameter declaration (on the left side of the `->` operator) must be enclosed in parentheses. Here is another example, involving two parameters:

```
(int a, int b) -> a + b;
```

### Using Multiple Statements in the Body of a Lambda Expression

You can write multiple statements in the body of a lambda expression, but if you do, you must enclose the statements in a set of curly braces, and you must write a `return` statement if the expression returns a value. Here is an example:

```
(int x) -> {
     int a = x * 2;
     return a;
};
```

### Accessing Variables Within a Lambda Expression

A lambda expression can access variables that are declared in the enclosing scope, as long as those variables are `final`, or effectively `final`. An *effectively* `final` variable is a variable whose value is never changed, but it isn't declared with the `final` key word.

In Code Listing 10-46, the `main` method uses a lambda expression that accesses a `final` variable named `factor` that is local to the `main` method.

**Code Listing 10-46** (`LambdDemo2.java`)

```
 1 import java.util.Scanner;
 2
 3 /**
 4    This program demonstrates a lambda expression
 5    that uses a final local variable.
 6 */
 7
 8 public class LambdaDemo2
 9 {
10    public static void main(String[] args)
11    {
12       final int factor = 10;
13       int num;
14
15       // Create a Scanner object for keyboard input.
16       Scanner keyboard = new Scanner(System.in);
17
18       // Create an object that implements IntCalculator.
19       IntCalculator multiplier = x -> x * factor;
20
21       // Get a number from the user.
22       System.out.print("Enter an integer number: ");
23       num = keyboard.nextInt();
24
25       // Display the number multiplied by 10.
26       System.out.println("Multiplied by 10, that number is " +
27                          multiplier.calculate(num));
28    }
29 }
```

**Program Output with Example Input Shown in Bold**

```
Enter an integer number: 10 [Enter]
Multiplied by 10, that number is 100
```

In Code Listing 10-46 we could remove the `final` key word from the variable declaration in line 12, and the program would still compile and execute correctly. This is because the `factor` variable is never modified, and therefore is effectively `final`.

## 10.12 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Attempting to access a private superclass member directly from a subclass.** Private superclass members cannot be directly accessed by a method in a subclass. The subclass must call a public or protected superclass method in order to access the superclass's private members.
- **Forgetting to call a superclass constructor explicitly when the superclass does not have a default constructor or a programmer-defined no-arg constructor.** When a superclass does not have a default constructor or a programmer-defined no-arg constructor, the subclass's constructor must explicitly call one of the constructors that the superclass does have.
- **Allowing the superclass's no-arg constructor to be implicitly called when you intend to call another superclass constructor.** If a subclass's constructor does not explicitly call a superclass constructor, Java automatically calls the superclass's no-arg constructor.
- **Forgetting to precede a call to an overridden superclass method with `super`.** When a subclass method calls an overridden superclass method, it must precede the method call with the key word `super` and a dot (`.`). Failing to do so results in the subclass's version of the method being called.
- **Forgetting a class member's access specifier.** When you do not give a class member an access specifier, it is granted package access by default. This means that any method in the same package may access the member.
- **Writing a body for an abstract method.** An abstract method cannot have a body. It must be overridden in a subclass.
- **Forgetting to terminate an abstract method's header with a semicolon.** An abstract method header does not have a body, and it must be terminated with a semicolon.
- **Failing to override an abstract method.** An abstract method must be overridden in a subclass.
- **Overloading an abstract method instead of overriding it.** Overloading is not the same as overriding. When a superclass has an abstract method, the subclass must have a method with the same signature as the abstract method.
- **Trying to instantiate an abstract class.** You cannot create an instance of an abstract class.
- **Implementing an interface but forgetting to provide all of the methods specified by the interface.** When a class implements an interface, all of the methods specified by the interface must be provided in the class.
- **Writing a method specified by an interface but failing to use the exact signature and return type.** When a class implements an interface, the class must have methods with the same signature and return type as the methods specified in the interface.

## Review Questions and Exercises

### Multiple Choice and True/False

1. In an inheritance relationship, this is the general class.
    a. subclass
    b. superclass
    c. slave class
    d. child class
2. In an inheritance relationship, this is the specialized class.
    a. superclass
    b. master class
    c. subclass
    d. parent class
3. This key word indicates that a class inherits from another class.
    a. `derived`
    b. `specialized`
    c. `based`
    d. `extends`
4. A subclass does not have access to these superclass members.
    a. public
    b. private
    c. protected
    d. all of these
5. This key word refers to an object's superclass.
    a. `super`
    b. `base`
    c. `superclass`
    d. `this`
6. In a subclass constructor, a call to the superclass constructor must __________.
    a. appear as the very first statement
    b. appear as the very last statement
    c. appear between the constructor's header and the opening brace
    d. not appear
7. The following is an explicit call to the superclass's default constructor.
    a. `default();`
    b. `class();`
    c. `super();`
    d. `base();`
8. A method in a subclass that has the same signature as a method in the superclass is an example of __________.
    a. overloading
    b. overriding

c. composition
d. an error

9. A method in a subclass having the same name as a method in the superclass but a different signature is an example of __________.
a. overloading
b. overriding
c. composition
d. an error

10. These superclass members are accessible to subclasses and classes in the same package.
a. private
b. public
c. protected
d. all of these

11. All classes directly or indirectly inherit from this class.
a. `Object`
b. `Super`
c. `Root`
d. `Java`

12. With this type of binding, the Java Virtual Machine determines at runtime which method to call, depending on the type of the object that a variable references.
a. static
b. early
c. flexible
d. dynamic

13. This operator can be used to determine whether a reference variable references an object of a particular class.
a. `isclass`
b. `typeof`
c. `instanceof`
d. `isinstance`

14. When a class implements an interface, it must __________.
a. overload all of the methods listed in the interface
b. provide all of the nondefault methods that are listed in the interface, with the exact signatures and return types specified
c. not have a constructor
d. be an abstract class

15. Fields in an interface are __________.
a. `final`
b. `static`
c. both `final` and `static`
d. not allowed

16. Abstract methods must be __________.
    a. overridden
    b. overloaded
    c. deleted and replaced with real methods
    d. declared as private
17. Abstract classes cannot __________.
    a. be used as superclasses
    b. have abstract methods
    c. be instantiated
    d. have fields
18. You use the __________ operator to define an anonymous inner class.
    a. `class`
    b. `inner`
    c. `new`
    d. `anonymous`
19. An anonymous inner class must __________.
    a. be a superclass
    b. implement an interface
    c. extend a superclass
    d. either b or c.
20. A functional interface is an interface with __________.
    a. only one abstract method.
    b. no abstract methods.
    c. only private methods.
    d. no name.
21. You can use a lambda expression to instantiate an object that __________.
    a. that has no constructor.
    b. extends any superclass.
    c. implements a functional interface
    d. does not implement an interface.
22. **True or False:** Constructors are not inherited.
23. **True or False:** In a subclass, a call to the superclass constructor can only be written in the subclass constructor.
24. **True or False:** If a subclass constructor does not explicitly call a superclass constructor, Java will not call any of the superclass's constructors.
25. **True or False:** An object of a superclass can access members declared in a subclass.
26. **True or False:** The superclass constructor always executes before the subclass constructor.
27. **True or False:** When a method is declared with the `final` modifier, it must be overridden in a subclass.
28. **True or False:** A superclass has a member with package access. A class that is outside the superclass's package but inherits from the superclass can access the member.
29. **True or False:** A superclass reference variable can reference an object of a subclass that extends the superclass.

30. **True or False:** A subclass reference variable can reference an object of the superclass.
31. **True or False:** When a class contains an abstract method, the class cannot be instantiated.
32. **True or False:** A class may only implement one interface.
33. **True or False:** By default all members of an interface are public.

### Find the Error

Find the error in each of the following code segments:

1.

```
// Superclass
public class Vehicle
{
   (Member declarations . . .)
}
// Subclass
public class Car expands Vehicle
{
   (Member declarations . . .)
}
```

2.

```
// Superclass
public class Vehicle
{
   private double cost;
   (Other methods . . .)
}
// Subclass
public class Car extends Vehicle
{
   public Car(double c)
   {
      cost = c;
   }
}
```

3.

```
// Superclass
public class Vehicle
{
   private double cost;
   public Vehicle(double c)
   {
      cost = c;
   }
   (Other methods . . .)
}
// Subclass
public class Car extends Vehicle
```

```
{
   private int passengers;
   public Car(int p)
   {
       passengers = c;
   }
   (Other methods . . .)
}
```

4.
```
// Superclass
public class Vehicle
{
   public abstract double getMilesPerGallon();
   (Other methods . . .)
}
// Subclass
public class Car extends Vehicle
{
   private int mpg;
   public int getMilesPerGallon();
   {
      return mpg;
   }
   (Other methods . . .)
}
```

## Algorithm Workbench

1. Write the first line of the definition for a `Poodle` class. The class should extend the `Dog` class.
2. Look at the following code, which is the first line of a class definition:

   `public class Tiger extends Felis`

   In what order will the class constructors execute?
3. Write the declaration for class `B`. The class's members should be as follows:
   - `m`, an integer. This variable should not be accessible to code outside the class or to any class that extends class `B`.
   - `n`, an integer. This variable should be accessible only to classes that extend class `B` or are in the same package as class `B`.
   - `setM`, `getM`, `setN`, and `getN`. These are the mutator and accessor methods for the member variables `m` and `n`. These methods should be accessible to code outside the class.
   - `calc`. This is a public abstract method.

   Next, write the declaration for class `D`, which extends class `B`. The class's members should be as follows:
   - `q`, a `double`. This variable should not be accessible to code outside the class.
   - `r`, a `double`. This variable should be accessible to any class that extends class `D` or is in the same package.

- `setQ`, `getQ`, `setR`, and `getR`. These are the mutator and accessor methods for the member variables `q` and `r`. These methods should be accessible to code outside the class.
- `calc`, a public method that overrides the superclass's abstract `calc` method. This method should return the value of `q` times `r`.

4. Write the statement that calls a superclass constructor and passes the arguments `x`, `y`, and `z`.
5. A superclass has the following method:

```
public void setValue(int v)
{
    value = v;
}
```

   Write a statement that may appear in a subclass that calls this method, passing 10 as an argument.

6. A superclass has the following abstract method:

```
public abstract int getValue();
```

   Write an example of a `getValue` method that can appear in a subclass.

7. Write the first line of the definition for a `Stereo` class. The class should extend the `SoundSystem` class, and it should implement the `CDplayable`, `TunerPlayable`, and `CassettePlayable` interfaces.
8. Write an interface named `Nameable` that specifies the following methods:

```
public void setName(String n)
public String getName()
```

9. Look at the following interface:

```
public interface Computable
{
   double compute(double x);
}
```

   Write a statement that uses a lambda expression to create an object that implements the `Computable` interface. The object's name should be `half`. The `half` object's `compute` method should return the value of the `x` parameter divided by 2.

## Short Answer

1. What is an "is-a" relationship?
2. A program uses two classes: `Animal` and `Dog`. Which class is the superclass and which is the subclass?
3. What is the superclass and what is the subclass in the following line?

```
public class Pet extends Dog
```

4. What is the difference between a protected class member and a private class member?
5. Can a subclass ever directly access the private members of its superclass?
6. Which constructor is called first, that of the subclass or the superclass?
7. What is the difference between overriding a superclass method and overloading a superclass method?
8. Reference variables can be polymorphic. What does this mean?

9. When does dynamic binding take place?
10. What is an abstract method?
11. What is an abstract class?
12. What are the differences between an abstract class and an interface?
13. When you instantiate an anonymous inner class, the class must do one of two things. What are they?
14. What is a functional interface?
15. What is a lambda expression?

## Programming Challenges

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. `Employee` and `ProductionWorker` Classes

VideoNote
The `Employee` and `Productionworker` Classes Problem

Design a class named `Employee`. The class should keep the following information in fields:

- Employee name
- Employee number in the format XXX–L, where each X is a digit within the range 0–9 and the L is a letter within the range A–M.
- Hire date

Write one or more constructors and the appropriate accessor and mutator methods for the class.

Next, write a class named `ProductionWorker` that extends the `Employee` class. The `ProductionWorker` class should have fields to hold the following information:

- Shift (an integer)
- Hourly pay rate (a `double`)

The workday is divided into two shifts: day and night. The shift field will be an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

### 2. `ShiftSupervisor` Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that extends the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a field that holds the annual salary and a field that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

### 3. `TeamLeader` Class

In a particular factory, a team leader is an hourly paid production worker that leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that extends the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have fields for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

### 4. `Essay` Class

Design an `Essay` class that extends the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives for an essay. The student's essay score can be up to 100 and is determined in the following manner:

Grammar: 30 points
Spelling: 20 points
Correct length: 20 points
Content: 30 points

Demonstrate the class in a simple program.

### 5. Course Grades

In a course, a teacher gives the following tests and assignments:

- A **lab activity** that is observed by the teacher and assigned a numeric score.
- A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
- An **essay** that is assigned a numeric score.
- A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have a `GradedActivity` array named `grades` as a field. The array should have four elements, one for each of the assignments previously described. The class should have the following methods:

| | |
|---|---|
| `setLab`: | This method should accept a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` field should reference this object. |
| `setPassFailExam`: | This method should accept a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` field should reference this object. |
| `setEssay`: | This method should accept an `Essay` object as its argument. (See Programming Challenge 4 for the `Essay` class. If you have not completed Programming Challenge 4, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` field should reference this object. |
| `setFinalExam`: | This method should accept a `FinalExam` object as its argument. This object should already hold the student's score for the final exam. Element 3 of the `grades` field should reference this object. |
| `toString`: | This method should return a string that contains the numeric scores and grades for each element in the `grades` array. |

Demonstrate the class in a program.

### 6. `Analyzable` Interface

Modify the `CourseGrades` class you created in Programming Challenge 5 so it implements the following interface:

```
public interface Analyzable
{
   double getAverage();
   GradedActivity getHighest();
   GradedActivity getLowest();
}
```

The `getAverage` method should return the average of the numeric scores stored in the `grades` array. The `getHighest` method should return a reference to the element of the grades array that has the highest numeric score. The `getLowest` method should return a reference to the element of the grades array that has the lowest numeric score. Demonstrate the new methods in a complete program.

### 7. `Person` and `Customer` Classes

Design a class named `Person` with fields for holding a person's name, address, and telephone number. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields.

Next, design a class named `Customer`, which extends the `Person` class. The `Customer` class should have a field for a customer number and a `boolean` field indicating whether the customer wishes to be on a mailing list. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate an object of the `Customer` class in a simple program.

### 8. `PreferredCustomer` Class

A retail store has a preferred customer plan where customers can earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store as follows:

- When a preferred customer spends $500, he or she gets a 5 percent discount on all future purchases.
- When a preferred customer spends $1,000, he or she gets a 6 percent discount on all future purchases.
- When a preferred customer spends $1,500, he or she gets a 7 percent discount on all future purchases.
- When a preferred customer spends $2,000 or more, he or she gets a 10 percent discount on all future purchases.

Design a class named `PreferredCustomer`, which extends the `Customer` class you created in Programming Challenge 7. The `PreferredCustomer` class should have fields for the amount of the customer's purchases and the customer's discount level. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate the class in a simple program.

### 9. `BankAccount` and `SavingsAccount` Classes

Design an abstract class named `BankAccount` to hold the following data for a bank account:

- Balance
- Number of deposits this month

- Number of withdrawals
- Annual interest rate
- Monthly service charges

The class should have the following methods:

| | |
|---|---|
| Constructor: | The constructor should accept arguments for the balance and annual interest rate. |
| `deposit`: | A method that accepts an argument for the amount of the deposit. The method should add the argument to the account balance. It should also increment the variable holding the number of deposits. |
| `withdraw`: | A method that accepts an argument for the amount of the withdrawal. The method should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals. |
| `calcInterest`: | A method that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:<br>*Monthly Interest Rate* = (*Annual Interest Rate*/12)<br>*Monthly Interest* = *Balance* * *Monthly Interest Rate*<br>*Balance* = *Balance* + *Monthly Interest* |
| `monthlyProcess`: | A method that subtracts the monthly service charges from the balance, calls the `calcInterest` method, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero. |

Next, design a `SavingsAccount` class that extends the `BankAccount` class. The `SavingsAccount` class should have a status field to represent an active or inactive account. If the balance of a savings account falls below $25, it becomes inactive. (The `status` field could be a `boolean` variable.) No more withdrawals may be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following methods:

| | |
|---|---|
| `withdraw`: | A method that determines whether the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the superclass version of the method. |
| `deposit`: | A method that determines whether the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above $25, the account becomes active again. A deposit is then made by calling the superclass version of the method. |
| `monthlyProcess`: | Before the superclass method is called, this method checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of $1 for each withdrawal above 4 is added to the superclass field that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below $25, the account becomes inactive.) |

### 10. `Ship`, `CruiseShip`, and `CargoShip` Classes

Design a `Ship` class that the following members:

- A field for the name of the ship (a string).
- A field for the year that the ship was built (a string).
- A constructor and appropriate accessors and mutators.
- A `toString` method that displays the ship's name and the year it was built.

Design a `CruiseShip` class that extends the `Ship` class. The `CruiseShip` class should have the following members:

- A field for the maximum number of passengers (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that overrides the `toString` method in the base class. The `CruiseShip` class's `toString` method should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that extends the `Ship` class. The `CargoShip` class should have the following members:

- A field for the cargo capacity in tonnage (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that overrides the `toString` method in the base class. The `CargoShip` class's `toString` method should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has a `Ship` array. Assign various `Ship`, `CruiseShip`, and `CargoShip` objects to the array elements. The program should then step through the array, calling each object's `toString` method. (See Code Listing 10-25 as an example.)

# CHAPTER 11

# Exceptions and Advanced File I/O

**TOPICS**

11.1 Handling Exceptions
11.2 Throwing Exceptions
11.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization
11.4 Common Errors to Avoid

## 11.1 Handling Exceptions

**CONCEPT: An exception is an object that is generated as the result of an error or an unexpected event. To prevent exceptions from crashing your program, you must write code that detects and handles them.**

VideoNote Handling Exceptions

There are many error conditions that can occur while a Java application is running that will cause it to halt execution. By now you have probably experienced this many times. For example, look at the program in Code Listing 11-1. This program attempts to read beyond the bounds of an array.

**Code Listing 11-1** (BadArray.java)

```
/**
   This program causes an error and crashes.
*/

public class BadArray
{
   public static void main(String[] args)
   {
      // Create an array with 3 elements.
      int[] numbers = { 1, 2, 3 };

      // Attempt to read beyond the bounds
      // of the array.
      for (int i = 0; i <= 3; i++)
         System.out.println(numbers[i]);
   }
}
```

**Program Output**

```
1
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
         at BadArray.main(BadArray.java:15)
```

The `numbers` array in this program has only three elements, with the subscripts 0 though 2. The program crashes when it tries to read the element at `numbers[3]`, and displays an error message similar to that shown at the end of the program output. This message indicates that an exception occurred, and it gives some information about it. An *exception* is an object that is generated in memory as the result of an error or an unexpected event. When an exception is generated, it is said to have been "thrown." Unless an exception is detected by the application and dealt with, it causes the application to halt.

To detect that an exception has been thrown and prevent it from halting your application, Java allows you to create exception handlers. An *exception handler* is a section of code that gracefully responds to exceptions when they are thrown. The process of intercepting and responding to exceptions is called *exception handling*. If your code does not handle an exception when it is thrown, the *default exception handler* deals with it, as shown in Code Listing 11-1. The default exception handler prints an error message and crashes the program.

The error that caused the exception to be thrown in Code Listing 11-1 is easy to avoid. If the loop were written properly, it would not have tried to read outside the bounds of the array. Some errors, however, are caused by conditions that are outside the application and cannot be avoided. For example, suppose an application creates a file on the disk and the user deletes it. Later the application attempts to open the file to read from it, and because it does not exist, an error occurs. As a result, an exception is thrown.

## Exception Classes

As previously mentioned, an exception is an object. Exception objects are created from classes in the Java API. The API has an extensive hierarchy of exception classes. A small part of the hierarchy is shown in Figure 11-1.

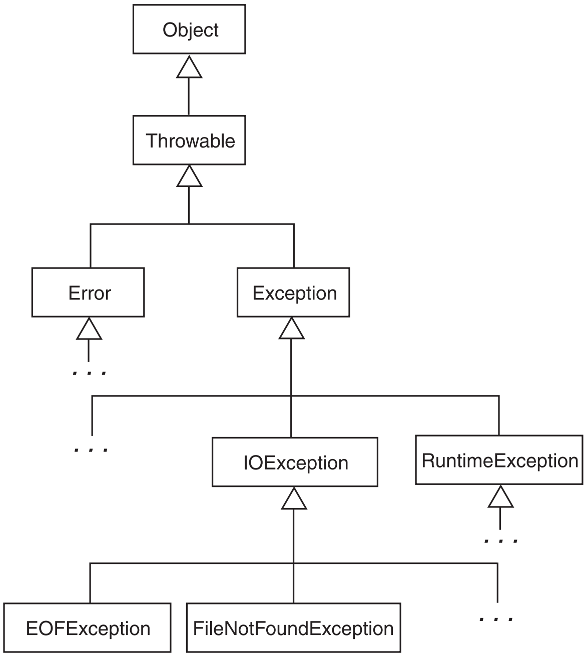
As you can see, all of the classes in the hierarchy inherit from the `Throwable` class. Just below the `Throwable` class are the classes `Error` and `Exception`. Classes that inherit from `Error` are for exceptions that are thrown when a critical error occurs, such as an internal error in the Java Virtual Machine or running out of memory. Your applications should not try to handle these errors because they are the result of a serious condition.

All of the exceptions that you will handle are instances of classes that inherit from `Exception`. Figure 11-1 shows two of these classes: `IOException` and `RuntimeException`. These classes also serve as superclasses. `IOException` serves as a superclass for exceptions that are related to input and output operations. `RuntimeException` serves as a superclass for exceptions that result from programming errors, such as an out-of-bounds array subscript.

The chart in Figure 11-1 shows two of the classes that inherit from the `IOException` class: `EOFException` and `FileNotFoundException`. These are examples of classes that exception objects are created from. An `EOFException` object is thrown when an application attempts to read beyond the end of a file, and a `FileNotFoundException` object is thrown when an application tries to open a file that does not exist.

**NOTE:** The exception classes are in packages in the Java API. For example, `FileNotFoundException` is in the `java.io` package. When you handle an exception that is not in the `java.lang` package, you will need the appropriate `import` statement.

**Figure 11-1** Part of the exception class hierarchy



## Handling an Exception

To handle an exception, you use a `try` statement. We will look at several variations of the `try` statement, beginning with the following general format:

```
try
{
   (try block statements . . . )
}
catch (ExceptionType parameterName)
{
   (catch block statements . . . )
}
```

First the key word `try` appears. Next, a block of code appears inside braces, which are required. This block of code is known as a *try block*. A *try block* is one or more statements that are executed and can potentially throw an exception. You can think of the code in the try block as being "protected" because the application will not halt if the try block throws an exception.

After the try block, a `catch` clause appears. A `catch` clause begins with the key word `catch`, followed by the code `(ExceptionType parameterName)`. This is a parameter variable declaration, where `ExceptionType` is the name of an exception class and `parameterName` is a variable name. If code in the try block throws an exception of the `ExceptionType` class, then the parameter variable will reference the exception object. In addition, the code that immediately follows the `catch` clause is executed. The code that immediately follows the `catch` clause is known as a `catch block`. Once again, the braces are required.

Let's look at an example of code that uses a `try` statement. The statement inside the following try block attempts to open the file *MyFile.txt*. If the file does not exist, the `Scanner` object throws an exception of the `FileNotFoundException` class. This code is designed to handle that exception if it is thrown.

```
try
{
   File file = new File("MyFile.txt");
   Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");

}
```

Let's look closer. First, the code in the try block is executed. If this code throws an exception, the Java Virtual Machine searches for a `catch` clause that can deal with the exception. In order for a `catch` clause to be able to deal with an exception, its parameter must be of a type that is compatible with the exception's type. Here is this code's `catch` clause:

```
catch (FileNotFoundException e)
```

This `catch` clause declares a reference variable named `e` as its parameter. The `e` variable can reference an object of the `FileNotFoundException` class. So, this `catch` clause can deal with an exception of the `FileNotFoundException` class. If the code in the try block throws an exception of the `FileNotFoundException` class, the `e` variable will reference the exception object and the code in the catch block will execute. In this case, the message "File not found." will be printed. After the catch block is executed, the program will resume with the code that appears after the entire `try/catch` construct.

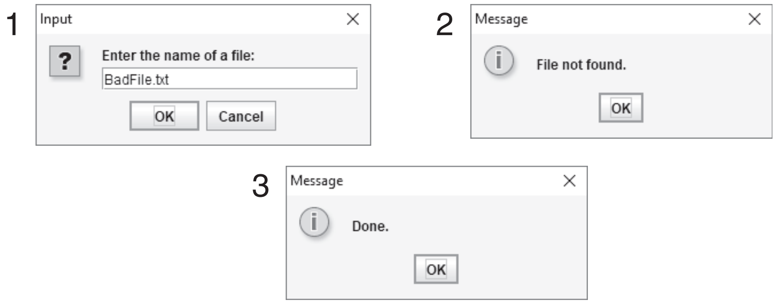
**NOTE:** The Java API documentation lists all of the exceptions that can be thrown from each method.

Code Listing 11-2 shows a program that asks the user to enter a file name, then attempts to open the file. If the file does not exist, an error message is printed. Figures 11-2 and 11-3 show examples of interaction with the program.

**Code Listing 11-2** (OpenFile.java)

```
import java.io.*;    // For File class and FileNotFoundException
import java.util.Scanner;          // For the Scanner class
import javax.swing.JOptionPane;    // For the JOptionPane class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class OpenFile
{
   public static void main(String[] args)
   {
      File file;                  // For file input
      Scanner inputFile;          // For file input
      String fileName;            // To hold a file name

      // Get a file name from the user.
      fileName = JOptionPane.showInputDialog("Enter " +
                           "the name of a file:");

      // Attempt to open the file.
      try
      {
         file = new File(fileName);
         inputFile = new Scanner(file);
         JOptionPane.showMessageDialog(null,
                           "The file was found.");
      }
      catch (FileNotFoundException e)
      {
         JOptionPane.showMessageDialog(null,
                           "File not found.");
      }
35
36         JOptionPane.showMessageDialog(null, "Done.");
37         System.exit(0);
38     }
39 }
```

**Figure 11-2** Interaction with the `OpenFile.java` program (assume that *BadFile.txt* does not exist) (Oracle Corporate Counsel)

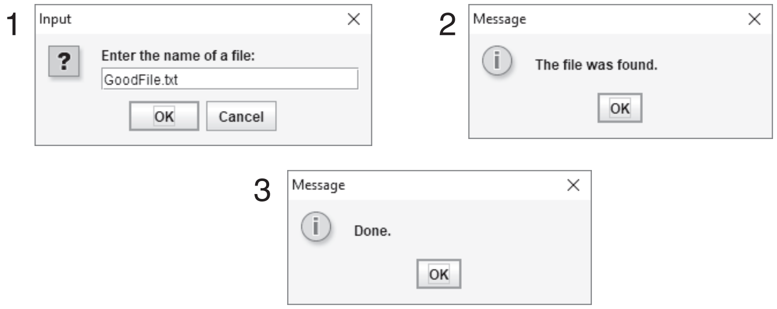


1 Enter the name of a file: BadFile.txt

2 File not found.

3 Done.

**Figure 11-3** Interaction with the `OpenFile.java` program (assume that *GoodFile.txt* does exist) (Oracle Corporate Counsel)



1 Enter the name of a file: GoodFile.txt

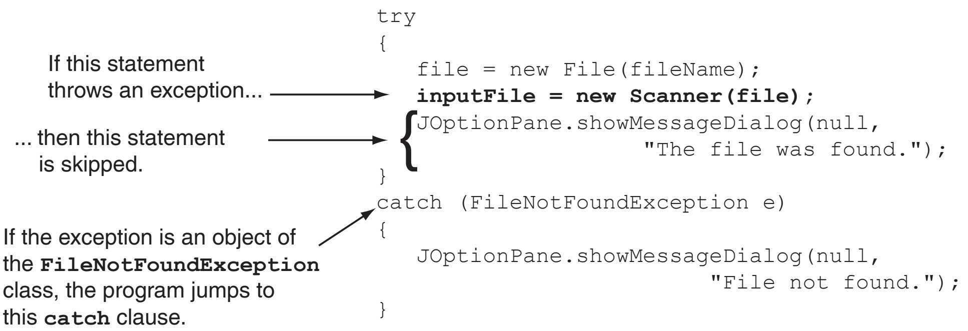
2 The file was found.

3 Done.

Look at the example run of the program in Figure 11-2. The user entered *BadFile.txt* as the file name. In line 25, the first statement inside the try block, a `File` object is created and this name is passed to the `File` constructor. In line 26 a reference to the `File` object is passed to the `Scanner` constructor. Because *BadFile.txt* does not exist, an exception of the `FileNotFoundException` class is thrown by the `Scanner` class constructor. When the exception is thrown, the program immediately exits the try block, skipping the remaining statement in the block (lines 27 through 28). The program jumps to the `catch` clause in line 30, which has a `FileNotFoundException` parameter, and executes the catch block that follows it. Figure 11-4 illustrates this sequence of events.

Notice that after the catch block executes, the program resumes at the statement that immediately follows the `try/catch` construct. This statement, which is in line 36, displays the message "Done."

**Figure 11-4** Sequence of events with an exception



```
try
{
   file = new File(fileName);
   inputFile = new Scanner(file);
   JOptionPane.showMessageDialog(null,
                     "The file was found.");
}
catch (FileNotFoundException e)
{
   JOptionPane.showMessageDialog(null,
                          "File not found.");
}
```

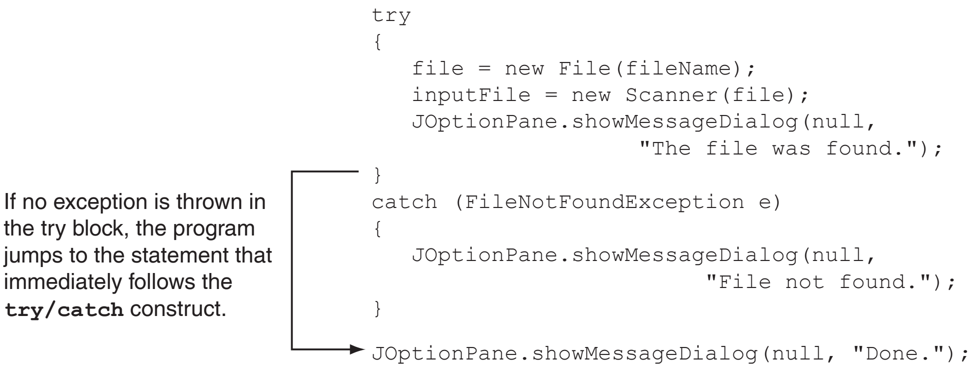
If this statement throws an exception...

... then this statement is skipped.

If the exception is an object of the **FileNotFoundException** class, the program jumps to this **catch** clause.

Now look at the example run of the program in Figure 11-3. In this case, the user entered *GoodFile.txt*, which is the name of a file that exists. No exception was thrown in the try block, so the program skips the `catch` clause and its catch block and jumps directly to the statement in line 36, which follows the `try/catch` construct. This statement displays the message "Done." Figure 11-5 illustrates this sequence of events.

**Figure 11-5** Sequence of events with no exception



```
try
{
   file = new File(fileName);
   inputFile = new Scanner(file);
   JOptionPane.showMessageDialog(null,
                     "The file was found.");
}
catch (FileNotFoundException e)
{
   JOptionPane.showMessageDialog(null,
                          "File not found.");
}
JOptionPane.showMessageDialog(null, "Done.");
```

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the **try/catch** construct.

## Retrieving the Default Error Message

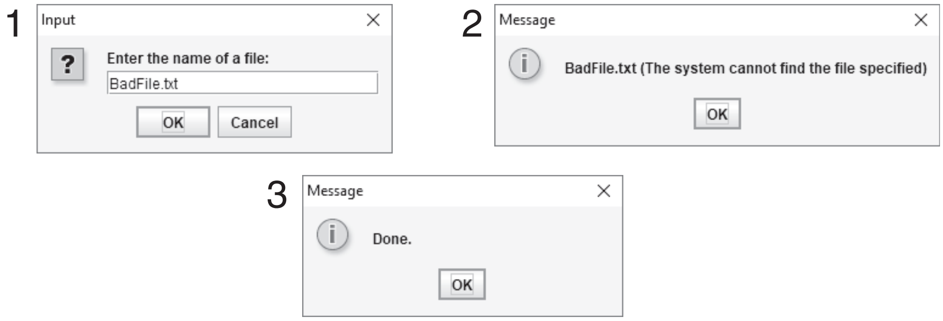
Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception. This is the same message that is displayed when the exception is not handled and the application halts. The program in Code Listing 11-3 demonstrates the `getMessage` method. This is a modified version of the program in Code Listing 11-2. Figure 11-6 shows the program running. In the figure, the user entered the name of a file that does not exist.

**Code Listing 11-3** (`ExceptionMessage.java`)

```
import java.io.*;                 // For file I/O classes
import java.util.Scanner;         // For the Scanner class
import javax.swing.JOptionPane;   // For the JOptionPane class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class ExceptionMessage
{
   public static void main(String[] args)
   {
      File file;                  // For file input
      Scanner inputFile;          // For file input
      String fileName;            // To hold a file name

      // Get a file name from the user.
      fileName = JOptionPane.showInputDialog("Enter " +
                                  "the name of a file:");

      // Attempt to open the file.
      try
      {
         file = new File(fileName);
         inputFile = new Scanner(file);
         JOptionPane.showMessageDialog(null,
                            "The file was found.");
      }
      catch (FileNotFoundException e)
      {
         JOptionPane.showMessageDialog(null, e.getMessage());
      }

      JOptionPane.showMessageDialog(null, "Done.");
      System.exit(0);
   }
}
```

Code Listing 11-4 shows another example. This program forces the `parseInt` method of the `Integer` wrapper class to throw an exception.

**Figure 11-6** Interaction with the `ExceptionMessage.java` program (assume that *BadFile.txt* does not exist) (Oracle Corporate Counsel)



1 Enter the name of a file: BadFile.txt

2 BadFile.txt (The system cannot find the file specified)

3 Done.

## Code Listing 11-4 (`ParseIntError.java`)

```
/**
   This program demonstrates how the Integer.parseInt
   method throws an exception.
*/

public class ParseIntError
{
   public static void main(String[] args)
   {
       String str = "abcde";
       int number;

       try
       {
          number = Integer.parseInt(str);
       }
       catch (NumberFormatException e)
       {
          System.out.println("Conversion error: " +
                             e.getMessage());
       }
   }
}
```

**Program Output**

```
Conversion error: For input string: "abcde"
```

The numeric wrapper classes' "parse" methods all throw an exception of the `NumberFormatException` type if the string being converted does not contain a convertible numeric value.

## Polymorphic References to Exceptions

Recall from Chapter 10 that a reference variable of a superclass type can reference subclass objects. This is called polymorphism. When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause. For example, all of the exceptions that we have dealt with inherit from the `Exception` class. So, a `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that inherits from the `Exception` class. For example, the `try` statement in Code Listing 11-4 could be written as follows:

```
try
{
   number = Integer.parseInt(str);
}
catch (Exception e)
{
   System.out.println("Conversion error: " +
                      e.getMessage());
}
```

Although the `Integer` class's `parseInt` method throws a `NumberFormatException` object, this code still works because the `NumberFormatException` class inherits from the `Exception` class.

## Using Multiple `catch` Clauses to Handle Multiple Exceptions

The programs we have studied so far test only for a single type of exception. In many cases, however, the code in the try block will be capable of throwing more than one type of exception. In such a case, you need to write a `catch` clause for each type of exception that could potentially be thrown.

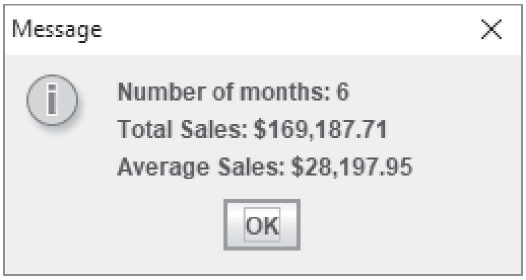
For example, the program in Code Listing 11-5 reads the contents of a file named *SalesData.txt*. Each line in the file contains the sales amount for one month, and the file has several lines. Here are the contents of the file:

```
24987.62
26978.97
32589.45
31978.47
22781.76
29871.44
```

The program in Code Listing 11-5 reads each number from the file and adds it to an accumulator variable. The try block contains code that can throw different types of exceptions.

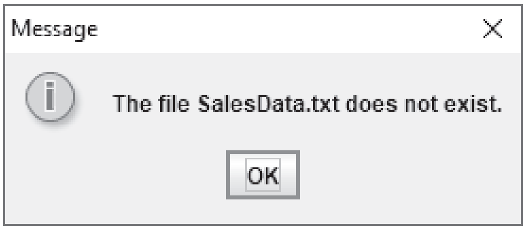
For example, the `Scanner` class's constructor can throw a `FileNotFoundException` if the file is not found, and the `Scanner` class's `nextDouble` method can throw an `InputMismatchException` (which is in the `java.util` package) if it reads a non-numeric value from the file. To handle these exceptions, the `try` statement has two `catch` clauses. Figure 11-7 shows the dialog box displayed by the program when no errors occur. This dialog box is displayed by the statement in lines 51 through 56. Figure 11-8 shows the dialog box displayed by the statement in lines 62 through 64 when the file cannot be found.

**Figure 11-7** Dialog box displayed by the `SalesReport.java` program when no error occurs (Oracle Corporate Counsel)



Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95

**Figure 11-8** Dialog box displayed by the `SalesReport.java` program when the file cannot be found (Oracle Corporate Counsel)



The file SalesData.txt does not exist.

**Code Listing 11-5** (`SalesReport.java`)

```
 1 import java.io.*;    // For File class and FileNotFoundException
 2 import java.util.*;  // For Scanner and InputMismatchException
 3 import javax.swing.JOptionPane; // For the JOptionPane class
 4
 5 /**
 6    This program demonstrates how multiple exceptions can
 7    be caught with one try statement.
 8 */
 9
10 public class SalesReport
11 {
12    public static void main(String[] args)
13    {
14       String filename = "SalesData.txt"; // File name
15       int months = 0;                    // Month counter
```

```
      double oneMonth;                  // One month's sales
      double totalSales = 0.0;          // Total sales
      double averageSales;              // Average sales

      try
      {
         // Open the file.
         File file = new File(filename);
         Scanner inputFile = new Scanner(file);

         // Process the contents of the file.
         while (inputFile.hasNext())
         {
            // Get a month's sales amount.
            oneMonth = inputFile.nextDouble();

            // Accumulate the amount.
            totalSales += oneMonth;

            // Increment the month counter
            months++;
         }

         // Close the file.
         inputFile.close();

         // Calculate the average.
         averageSales = totalSales / months;

         // Display the results.
         JOptionPane.showMessageDialog(null,
            String.format("Number of months: %d\n" +
                          "Total Sales: $%,.2f\n" +
                          "Average Sales: $%,.2f",
                          months, totalSales, averageSales));
      }
      catch(FileNotFoundException e)
      {
         // Thrown by the Scanner constructor when
         // the file is not found.
         JOptionPane.showMessageDialog(null,
              "The file " + filename + " does not exist.");
      }
      catch(InputMismatchException e)
      {
         // Thrown by the Scanner class's nextDouble
         // method when a non-numeric value is found.
```

```
            JOptionPane.showMessageDialog(null,
               "Non-numeric data found in the file.");
         }

         System.exit(0);
      }
}
```

When an exception is thrown by code in the try block, the JVM begins searching the `try` statement for a `catch` clause that can handle it. It searches the `catch` clauses from top to bottom and passes control of the program to the first `catch` clause with a parameter that is compatible with the exception.

### Using Exception Handlers to Recover from Errors

The program in Code Listing 11-5 demonstrates how a `try` statement can have several `catch` clauses in order to handle different types of exceptions. However, the program does not use the exception handlers to recover from any of the errors. Regardless of whether the file is not found or a non-numeric item is encountered in the file, this program still halts. The program in Code Listing 11-6 is a better example of effective exception handling. It attempts to recover from as many of the exceptions as possible.

**Code Listing 11-6** (`SalesReport2.java`)

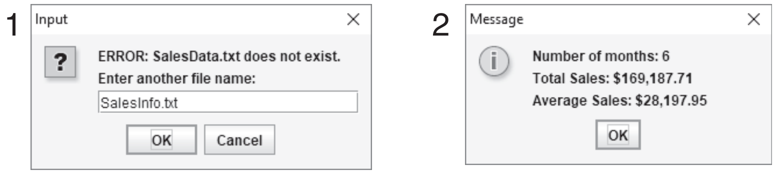
```
import java.io.*;    // For File class and FileNotFoundException
import java.util.*;  // For Scanner and InputMismatchException
import javax.swing.JOptionPane; // For the JOptionPane class

/**
   This program demonstrates how exception handlers can
   be used to recover from errors.
*/

public class SalesReport2
{
   public static void main(String[] args)
   {
      String filename = "SalesData.txt"; // File name
      int months = 0;                    // Month counter
      double oneMonth;                   // One month's sales
      double totalSales = 0.0;           // Total sales
      double averageSales;               // Average sales

      // Attempt to open the file by calling the
      // openfile method.
      Scanner inputFile = openFile(filename);

      // If the openFile method returned null, then
      // the file was not found. Get a new file name.
      while (inputFile == null)
      {
         filename = JOptionPane.showInputDialog(
                        "ERROR: " + filename +
                        " does not exist.\n" +
                        "Enter another file name: ");
         inputFile = openFile(filename);
      }

      // Process the contents of the file.
      while (inputFile.hasNext())
      {
         try
         {
            // Get a month's sales amount.
            oneMonth = inputFile.nextDouble();

            // Accumulate the amount.
            totalSales += oneMonth;

            // Increment the month counter.
            months++;
         }
         catch(InputMismatchException e)
         {
            // Display an error message.
            JOptionPane.showMessageDialog(null,
               "Non-numeric data found in the file.\n" +
               "The invalid record will be skipped.");

            // Skip past the invalid data.
            inputFile.nextLine();
         }
      }

      // Close the file.
      inputFile.close();

      // Calculate the average.
      averageSales = totalSales / months;

      // Display the results.
      JOptionPane.showMessageDialog(null,
         String.format("Number of months: %d\n" +
                       "Total Sales: $%,.2f\n" +
 71                        "Average Sales: $%,.2f",
 72                        months, totalSales, averageSales));
 73
 74       System.exit(0);
 75    }
 76
 77    /**
 78       The opeFile method opens the specified file and
 79       returns a reference to a Scanner object.
 80       @param filename The name of the file to open.
 81       @return A Scanner reference, if the file exists
 82               Otherwise, null is returned.
 83    */
 84
 85    public static Scanner openFile(String filename)
 86    {
 87       Scanner scan;
 88
 89       // Attempt to open the file.
 90       try
 91       {
 92          File file = new File(filename);
 93          scan = new Scanner(file);
 94       }
 95       catch(FileNotFoundException e)
 96       {
 97          scan = null;
 98       }
 99
100       return scan;
101    }
102 }
```

Let's look at how this program recovers from a `FileNotFoundException`. The `openFile` method, in lines 85 through 101, accepts a file name as its argument. The method creates a `File` object (passing the file name to the constructor) and a `Scanner` object. If the `Scanner` class constructor throws a `FileNotFoundException`, the method returns `null`. Otherwise, it returns a reference to the `Scanner` object. In the `main` method, a loop is used in lines 26 through 33 to ask the user for a different file name in the event that the `openFile` method returns `null`.

Now let's look at how the program recovers from unexpectedly encountering a non-numeric item in the file. The statement in line 41, which calls the `Scanner` class's `nextDouble` method, is wrapped in a `try` statement that catches the `InputMismatchException`. If this exception is thrown by the `nextDouble` method, the catch block in lines 49 through 58 displays a message indicating that a non-numeric item was encountered and that the invalid record will be skipped. The invalid data is then read from the file with the `nextLine` method in line 57. Because the statement `months++` in line 47 is in the try block, it will not be executed when the exception occurs, so the number of months will still be correct. The loop continues processing with the next line in the file.

Let's look at some examples of how the program recovers from these errors. Suppose we rename *SalesData.txt* file as *SalesInfo.txt*. Figure 11-9 shows an example running of the program.

**Figure 11-9** Interaction with the `SalesReport2.java` program (Oracle Corporate Counsel)



1
ERROR: SalesData.txt does not exist.
Enter another file name:
SalesInfo.txt

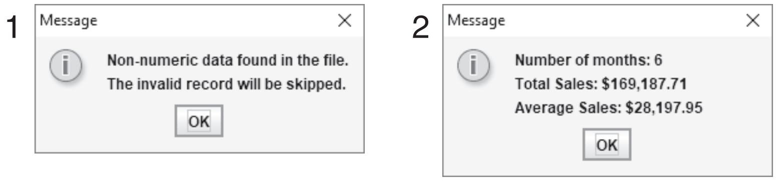
2
Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95

Now, suppose we change the name of the file back to *SalesData.txt* and edit its contents as follows:

```
24987.62
26978.97
abc
31978.47
22781.76
29871.44
```

Notice that the third item is no longer a number. Figure 11-10 shows an example running of the program.

**Figure 11-10** Dialog boxes displayed by the `SalesReport2.java` program (Oracle Corporate Counsel)



1
Non-numeric data found in the file.
The invalid record will be skipped.

2
Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95

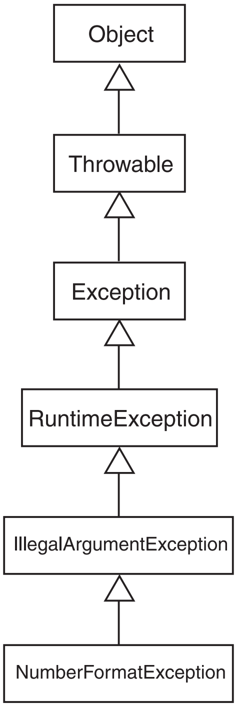
### Handle Each Exception Only Once in a `try` Statement

Not including polymorphic references, a `try` statement may have only one `catch` clause for each specific type of exception. For example, the following `try` statement will cause the compiler to issue an error message because it handles a `NumberFormatException` object with two `catch` clauses:

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println("Bad number format.");
}
// ERROR!!! NumberFormatException has already been caught!
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
```

Sometimes you can cause this error by using polymorphic references. For example, look at Figure 11-11, which shows an inheritance hierarchy for the `NumberFormatException` class.

**Figure 11-11** Inheritance hierarchy for the `NumberFormatException` class


Object
Throwable
Exception
RuntimeException
IllegalArgumentException
NumberFormatException

As you can see from the figure, the `NumberFormatException` class inherits from the `IllegalArgumentException` class. Now look at the following code:

```
try
{
   number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
// This will also cause an error.
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
```

The compiler issues an error message regarding the second `catch` clause, reporting that `NumberFormatException` has already been caught. This is because the first `catch` clause, which catches `IllegalArgumentException` objects, will polymorphically catch `NumberFormatException` objects.

If you are handling multiple exceptions in the same `try` statement and some of the exceptions are related to each other through inheritance, then you should handle the more specialized exception classes before the more general exception classes. We can rewrite the previous code as follows, with no errors:

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
```

## The `finally` Clause

The `try` statement may have an optional `finally` clause, which must appear after all of the `catch` clauses. Here is the general format of a `try` statement with a `finally` clause:

```
try
{
   (try block statements . . . )
}
```

```
catch (ExceptionType ParameterName)
{
    (catch block statements . . . )
}
finally
{
    (finally block statements . . . )
}
```

The *finally block* is one or more statements that are always executed after the try block has executed and after any catch blocks have executed if an exception was thrown. The statements in the finally block execute whether an exception occurs or not. For example, the following code opens a file of `doubles` and reads its contents. The outer `try` statement opens the file and has a `catch` clause that catches the `FileNotFoundException`. The inner `try` statement reads values from the file and has a `catch` clause that catches the `InputMismatchException`. The finally block closes the file regardless of whether an `InputMismatchException` occurs.

```
try
{
   // Open the file.
   File file = new File(filename);
   Scanner inputFile = new Scanner(file);

   try
   {
      // Read and display the file's contents.
      while (inputFile.hasNext())
      {
         System.out.println(inputFile.nextDouble());
      }
   }
   catch (InputMismatchException e)
   {
      System.out.println("Invalid data found.");
   }
   finally
   {
      // Close the file.
      inputFile.close();
   }
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
```

## The Stack Trace

Quite often, a method will call another method, which will call yet another method. For example, method A calls method B, which calls method C. The *call stack* is an internal list of all the methods that are currently executing.

When an exception is thrown by a method that is executing under several layers of method calls, it is sometimes helpful to know which methods were responsible for the method being called. A *stack trace* is a list of all the methods in the call stack. It indicates the method that was executing when an exception occurred and all of the methods that were called in order to execute that method. For example, look at the program in Code Listing 11-7. It has three methods: `main`, `myMethod`, and `produceError`. The `main` method calls `myMethod`, which calls `produceError`. The `produceError` method causes an exception by passing an invalid position number to the `String` class's `charAt` method. The exception is not handled by the program, but is dealt with by the default exception handler.

**Code Listing 11-7** `(StackTrace.java)`

```
/**
   This program demonstrates the stack trace that is
   produced when an exception is thrown.
*/

public class StackTrace
{
   public static void main(String[] args)
   {
      System.out.println("Calling myMethod...");
      myMethod();
      System.out.println("Method main is done.");
   }

   /**
      MyMethod
   */

   public static void myMethod()
   {
      System.out.println("Calling produceError...");
      produceError();
      System.out.println("myMethod is done.");
   }

   /**
      produceError
   */

   public static void produceError()
   {
      String str = "abc";

      // The following statement will cause an error.
      System.out.println(str.charAt(3));
      System.out.println("produceError is done.");
   }
}
```

**Program Output**

```
Calling myMethod...
Calling produceError...
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
      String index out of range: 3
        at java.lang.String.charAt(Unknown Source)
        at StackTrace.produceError(StackTrace.java:35)
        at StackTrace.myMethod(StackTrace.java:22)
        at StackTrace.main(StackTrace.java:11)
```

When the exception occurs, the error message shows a stack trace listing the methods that were called in order to produce the exception. The first method that is listed in the stack trace, `charAt`, is the method that is responsible for the exception. The next method, `produceError`, is the method that called `charAt`. The next method, `myMethod`, is the method that called `produceError`. The last method, `main`, is the method that called `myMethod`. The stack trace shows the chain of methods that were called when the exception was thrown.

**NOTE:** All exception objects have a `printStackTrace` method, inherited from the `Throwable` class, which can be used to print a stack trace.

## Handling Multiple Exceptions with One `catch` Clause

In versions of Java prior to Java 7, each `catch` clause can handle only one type of exception. Beginning with Java 7, however, a `catch` clause can handle more than one type of exception. This can reduce a lot of duplicated code in a `try` statement that needs to catch multiple exceptions, but perform the same operation for each one. For example, suppose we have the following `try` statement in a program:

```
try
{
   (try block statements . . . )
}
catch(NumberFormatException ex)
{
   respondToError();
}
```

```
catch(IOException ex)
{
   respondToError();
}
```

This `try` statement has two `catch` clauses: one that handles a `NumberFormatException`, and another that handles an `IOException`. Notice that both catch blocks do the same thing: they call a method named `respondToError`. Because both catch blocks perform the same operation, the `catch` clauses can be combined into a single `catch` clause that handles both types of exception, as shown here:

```
try
{
   (try block statements . . . )
}
catch(NumberFormatException | IOException ex)
{
   respondToError();
}
```

Notice in the `catch` clause that the exception types are separated by a `|` symbol, which is the same symbol as that used for the logical OR operator. You can think of this as meaning that the clause will catch a `NumberFormatException` or an `IOException`. The following code shows a `catch` clause that handles three types of exceptions:

```
try
{
   (try block statements . . . )
}
catch(NumberFormatException | IOException | InputMismatchException ex)
{
   respondToError();
}
```

In this code, the `catch` clause will handle a `NumberFormatException` or an `IOException` or an `InputMismatchException`.

The ability to catch multiple types of exceptions with a single `catch` clause is known as *multi-catch*, and was introduced in Java 7. Code Listing 11-8 shows a complete program that uses multi-catch. The `catch` clause in line 34 can handle a `FileNotFoundException` or an `InputMismatchException`.

**Code Listing 11-8** `(MultiCatch.java)`

```
1 import java.io.*;     // For File class and FileNotFoundException
2 import java.util.*;   // For Scanner and InputMismatchException
3
4 /**
5    This program demonstrates how multiple exceptions can
6    be caught with a single catch clause.
7 */
8
```

```
public class MultiCatch
{
   public static void main(String[] args)
   {
      int number;    // To hold a number from the file

      try
      {
         // Open the file.
         File file = new File("Numbers.txt");
         Scanner inputFile = new Scanner(file);

         // Process the contents of the file.
         while (inputFile.hasNext())
         {
            // Get a number from the file.
            number = inputFile.nextInt();

            // Display the number.
            System.out.println(number);
         }

         // Close the file.
         inputFile.close();
      }
      catch(FileNotFoundException | InputMismatchException ex)
      {
         // Display an error message.
         System.out.println("Error processing the file.");
      }
   }
}
```

**NOTE:** If you are using a version of Java prior to Java 7, you cannot use multi-catch.

## When an Exception Is Not Caught

When an exception is thrown, it cannot be ignored. It must be handled by the program, or by the default exception handler. When the code in a method throws an exception, the normal execution of that method stops and the JVM searches for a compatible exception handler inside the method. If there is no code inside the method to handle the exception, then control of the program is passed to the previous method in the call stack (that is, the method that called the offending method). If that method cannot handle the exception, then control is passed again, up the call stack, to the previous method. This continues until control reaches the `main` method. If the `main` method does not handle the exception, then the program is halted and the default exception handler handles the exception.

This was the case for the program in Code Listing 11-7. Because the `produceError` method did not handle the exception, control was passed back to `myMethod`. It didn't handle the exception either, so control was passed back to `main`. Because `main` didn't handle the exception, the program halted and the default exception handler displayed the error messages.

## Checked and Unchecked Exceptions

In Java, there are two categories of exceptions: unchecked and checked. *Unchecked exceptions* are those that inherit from the `Error` class or the `RuntimeException` class. Recall that the exceptions that inherit from `Error` are thrown when a critical error occurs, such as running out of memory. You should not handle these exceptions because the conditions that cause them can rarely be dealt with in the program. Also recall that `RuntimeException` serves as a superclass for exceptions that result from programming errors, such as an out-of-bounds array subscript. It is best not to handle these exceptions either, because they can be avoided with properly written code. So, you should not handle unchecked exceptions.

All of the remaining exceptions (that is, those that do *not* inherit from `Error` or `RuntimeException`) are *checked exceptions*. These are the exceptions that you should handle in your program. If the code in a method can potentially throw a checked exception, then that method must meet one of the following requirements:

- It must handle the exception, or
- It must have a `throws` clause listed in the method header.

The `throws` clause informs the compiler of the exceptions that could get thrown from a method. For example, look at the following method:

```
// This method will not compile!
public void displayFile(String name)
{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);

   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
      System.out.println(inputFile.nextLine());
   }

   // Close the file.
   inputFile.close();
}
```

The code in this method is capable of throwing a `FileNotFoundException`, which is a checked exception. Because the method does not handle this exception, it must have a `throws` clause in its header or it will not compile.

The key word `throws` is written at the end of the method header, followed by a list of the types of exceptions that the method can throw. Here is the revised method header:

```
public void displayFile(String name) throws FileNotFoundException
```

The `throws` clause tells the compiler that this method can throw a `FileNotFoundException`. (If there is more than one type of exception, you separate them with commas.)

Now you know why you wrote a `throws` clause on methods that perform file operations in the previous chapters. We did not handle any of the checked exceptions that might occur, so we had to inform the compiler that our methods might pass them up the call stack.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

11.1 Briefly describe what an exception is.

11.2 What does it mean to “throw" an exception?

11.3 If an exception is thrown and the program does not handle it, what happens?

11.4 Other than the `Object` class, what is the superclass for all exceptions?

11.5 What is the difference between exceptions that inherit from the `Error` class and exceptions that inherit from the `Exception` class?

11.6 What is the difference between a try block and a catch block?

11.7 After the catch block has handled the exception, where does program execution resume?

11.8 How do you retrieve an error message from an exception?

11.9 If multiple exceptions can be thrown by code in a try block, how does the JVM know which `catch` clause it should pass the control of the program to?

11.10 When does the code in a finally block execute?

11.11 What is the call stack? What is a stack trace?

11.12 A program’s `main` method calls method A, which calls method B. None of these methods performs any exception handling. The code in method B throws an exception. Describe what happens.

11.13 What are the differences between a checked and an unchecked exception?

11.14 When are you required to have a `throws` clause in a method header?

## 11.2 Throwing Exceptions

**CONCEPT: You can write code that throws one of the standard Java exceptions, or an instance of a custom exception class that you have designed.**

You can use the `throw` statement to throw an exception manually. The general format of the `throw` statement is as follows:

```
throw new ExceptionType(MessageString);
```

The `throw` statement causes an exception object to be created and thrown. In this general format, *`ExceptionType`* is an exception class name and *`MessageString`* is an optional `String` argument passed to the exception object’s constructor. The *`MessageString`* argument contains a custom error message that can be retrieved from the exception object’s `getMessage` method. If you do not pass a message to the constructor, the exception will have a `null` message. Here is an example of a `throw` statement:

```
throw new Exception("Out of fuel");
```

This statement creates an object of the `Exception` class and passes the string “Out of fuel” to the object’s constructor. The object is then thrown, which causes the exception-handling process to begin.

**NOTE:** Don’t confuse the `throw` statement with the `throws` clause. The `throw` statement causes an exception to be thrown. The `throws` clause informs the compiler that a method throws one or more exceptions.

Recall the `Die` class from Chapter 6. The class simulates a gaming die, and its constructor accepts an argument specifying the number of sides for the die. Suppose we want to make sure that the number of sides is not less than a minimum value (after all, it wouldn’t make sense to have a one-sided die, or a zero-sided die.)

One way to accomplish this is to have the constructor throw an exception when an invalid argument is passed. Code Listing 11-9 shows the modified code for the `Die` class:

**Code Listing 11-9** (`Die.java`)

```java
import java.util.Random;

/**
   The Die class simulates a six-sided die.
*/

public class Die
{
   private final int MIN_SIDES = 4;
   private int sides;   // Number of sides
   private int value;   // The die's value

   /**
      The constructor performs an initial
      roll of the die.
      @param numSides The number of sides for this die.
   */

   public Die(int numSides)
   {
      // Validate the number of sides.
      if (numSides < MIN_SIDES)
      {
         throw new IllegalArgumentException(
            "The die must have at least " +
            MIN_SIDES + " sides.");
      }
28
29       // Store the number of sides and roll.
30       sides = numSides;
31       roll();
32    }
33
34    /**
35       The roll method simulates the rolling of
36       the die.
37    */
38
39    public void roll()
40    {
41       // Create a Random object.
42       Random rand = new Random();
43
44       // Get a random value for the die.
45       value = rand.nextInt(sides) + 1;
46    }
47
48    /**
49       getSides method
50       @return The number of sides for this die.
51    */
52
53    public int getSides()
54    {
55       return sides;
56    }
57
58    /**
59       getValue method
60       @return The value of the die.
61    */
62
63    public int getValue()
64    {
65       return value;
66    }
67 }
```

Line 9 declares a `final` field named `MIN_SIDES`, initialized with the value 4. This is the minimum value that the class will accept for the number of sides. In the constructor, in line 22 we test the value of the `numSides` parameter, to determine whether it is less than `MIN_SIDES`. If so, lines 24 through 26 throw an `IllegalArgumentException`. The message

*The die must have at least 4 sides* is passed to the exception object's constructor. When we catch this exception, we can retrieve the message by calling the object's `getMessage` method. The `IllegalArgumentException` class was chosen for this error condition because it seems like the most appropriate exception to throw in response to an illegal argument being passed to the constructor. (Note that `IllegalArgumentException` inherits from `RuntimeException`, which inherits from `Exception`.)

> **NOTE:** Because the `IllegalArgumentException` class inherits from the `RuntimeException` class, it is unchecked. If we had chosen a checked exception class, we would have to put a throws clause in the constructor's header.

The program in Code Listing 11-10 demonstrates how the modified constructor works.

**Code Listing 11-10** (`DieExceptionDemo.java`)

```
/**
   This program demonstrates how the Die class throws
   an exception when an invalid value is passed to the
   constructor.
*/

public class DiceExceptionDemo
{
   public static void main(String[] args)
   {
      final int DIE_SIDES = 1;  // Number of sides

      // Create an instance of the Die class.
      Die die = new Die(DIE_SIDES);

      System.out.println("Initial value of the die:");
      System.out.println(die.getValue());
   }
}
```

**Program Output**

```
Exception in thread ”main” java.lang.IllegalArgumentException: The die must
have at least 4 sides.
     at Die.<init>(Die.java:24)
     at DiceExceptionDemo.main(DiceExceptionDemo.java:14)
```

## Creating Your Own Exception Classes

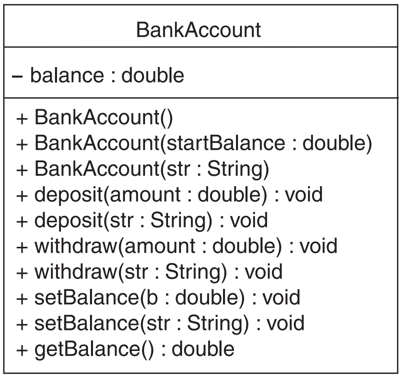
To meet the needs of a specific class or application, you can create your own exception classes by extending the `Exception` class or one of its subclasses.

Let's look at an example that uses programmer-defined exceptions. Recall the `BankAccount` class from Chapter 6. This class holds the data for a bank account. A UML diagram for the class is shown in Figure 11-12.

There are a number of errors that could cause a `BankAccount` object to perform its duties incorrectly. Here are some specific examples:

- A negative starting balance is passed to the constructor.
- A negative number is passed to the `deposit` method.
- A negative number is passed to the `withdraw` method.
- The amount passed to the `withdraw` method exceeds the account's balance.

**Figure 11-12** UML diagram for the BankAccount class


BankAccount
– balance : double
+ BankAccount()
+ BankAccount(startBalance : double)
+ BankAccount(str : String)
+ deposit(amount : double) : void
+ deposit(str : String) : void
+ withdraw(amount : double) : void
+ withdraw(str : String) : void
+ setBalance(b : double) : void
+ setBalance(str : String) : void
+ getBalance() : double

We can create our own exceptions that represent each of these error conditions. Then we can rewrite the class so it throws one of our custom exceptions when any of these errors occur. Let's start by creating an exception class for a negative starting balance. Code Listing 11-11 shows an exception class named `NegativeStartingBalance`.

**Code Listing 11-11** (`NegativeStartingBalance.java`)

```
/**
   NegativeStartingBalance exceptions are thrown by the
   BankAccount class when a negative starting balance is
   passed to the constructor.
*/

public class NegativeStartingBalance
                        extends Exception
{
   /**
      This constructor uses a generic
      error message.
   */

   public NegativeStartingBalance()
   {
      super("Error: Negative starting balance");
   }

   /**
      This constructor specifies the bad starting
      balance in the error message.
      @param The bad starting balance.
   */

   public NegativeStartingBalance(double amount)
   {
      super("Error: Negative starting balance: " +
            amount);
   }
}
```

Notice that this class extends the `Exception` class. It has two constructors. The no-arg constructor passes the string `"Error: Negative starting balance"` to the superclass constructor. This is the error message that is retrievable from an object's `getMessage` method. The second constructor accepts the starting balance as a `double` argument. This amount is used to pass a more detailed error message containing the starting balance amount to the superclass constructor.

The following code shows one of the `BankAccount` constructors rewritten to throw a `NegativeStartingBalance` exception when a negative value is passed as the starting balance.

```
public BankAccount(double startBalance)
                 throws NegativeStartingBalance
{
   if (startBalance < 0)
      throw new NegativeStartingBalance(startBalance);

   balance = startBalance;
}
```

Note that `NegativeStartingBalance` extends the `Exception` class. This means that it is a checked exception class. Because of this, the constructor header must have a `throws` clause listing the exception type.

You will find the modified *BankAccount.java* file in this chapter's source code, available on the book's companion Web site at www.pearsonhighered.com/gaddis. The program in Code Listing 11-12 demonstrates the new constructor by forcing it to throw the `NegativeStartingBalance` exception.

**Code Listing 11-12** (`AccountTest.java`)

```java
/**
   This program demonstrates how the BankAccount
   class constructor throws custom exceptions.
*/

public class AccountTest
{
   public static void main(String [] args)
   {
      // Force a NegativeStartingBalance exception.
      try
      {
         BankAccount account =
                      new BankAccount(-100.0);
      }
      catch(NegativeStartingBalance e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

**Program Output**

```
Error: Negative starting balance: -100.0
```

## Using the `@exception` Tag in Documentation Comments

When writing the documentation comments for a method, you can document the exceptions thrown by the method by using an `@exception` tag. When the `javadoc` utility sees an `@exception` tag inside a method's documentation comments, it knows that the name of an exception appears next, followed by a description of the events that cause the exception. The general format of an `@exception` tag comment is as follows:

`@exception` *`ExceptionName Description`*

*`ExceptionName`* is the name of an exception and *`Description`* is a description of the circumstances that cause the exception. Remember the following points about `@exception` tag comments:

- The `@exception` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

When a method's documentation comments contain an `@exception` tag, the `javadoc` utility will create a Throws section in the method's documentation. This is where the descriptions of the exceptions thrown by the method will be listed. As an example, here are the documentation comments for the `BankAccount` class's constructor presented earlier:

```
/**
   This constructor sets the starting balance
   to the value passed as an argument.
   @param startBalance The starting balance.
   @exception NegativeStartingBalance When
               startBalance is negative.
*/
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

11.15 What does the `throw` statement do?

11.16 What is the purpose of the argument that is passed to an exception object's constructor? What happens if you do not pass an argument to the constructor?

11.17 What is the difference between the `throw` statement and the `throws` clause?

11.18 If a method has a `throw` statement, does it always have to have a `throws` clause in its header? Why or why not?

11.19 If you are writing a custom exception class, how can you make sure it is checked? How can you make sure it is unchecked?

## 11.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization

**CONCEPT: A file that contains raw binary data is known as a binary file. The content of a binary file is not formatted as text, and not meant to be opened in a text editor. A random access file is a file that allows a program to read data from any location within the file, or write data to any location within the file. Object serialization is the process of converting an object to a series of bytes and saving them to a file. Deserialization is the process of reconstructing a serialized object.**

### Binary Files

All the files you've been working with so far have been text files. This means that the data stored in the files has been formatted as text. Even a number, when stored in a text file with the `print` or `println` method, is converted to text. For example, consider the following program segment:

```
PrintWriter outputFile = new PrintWriter("Number.txt");
int x = 1297;
outputFile.print(x);
```

The last statement writes the contents of the variable `x` to the *Number.txt* file. When the number is written, however, it is stored as the characters `'1'`, `'2'`, `'9'`, and `'7'`. This is illustrated in Figure 11-13.

**Figure 11-13** The number 1297 expressed as characters

1297 expressed as characters

| | | | |
|---|---|---|---|
| '1' | '2' | '9' | '7' |

When a number such as 1297 is stored in the computer's memory, it isn't stored as text, however. It is formatted as a binary number. Figure 11-14 shows how the number 1297 is stored in memory, in an `int` variable, using binary. Recall that `int` variables occupy four bytes.

**Figure 11-14** The number 1297 as a binary number, as it is stored in memory

1297 as a binary number

| | | | |
|---|---|---|---|
| 00000000 | 00000000 | 00000101 | 00010001 |

The binary representation of the number shown in Figure 11-14 is the way the raw data is stored in memory. In fact, this is sometimes called the *raw binary format*. Data can be stored in a file in its raw binary format. A file that contains binary data is often called a *binary file*.

Storing data in its binary format is more efficient than storing it as text because there are fewer conversions to take place. In addition, there are some types of data that should only be stored in their raw binary format. Images are an example. However, when data is stored in a binary file, you cannot open the file in a text editor such as Notepad. When a text editor opens a file, it assumes the file contains text.

### Writing Data to a Binary File

To write data to a binary file you must create objects from the following classes:

| | |
|---|---|
| `FileOutputStream` | This class, which is in the `java.io` package, allows you to open a file for writing binary data and establish a connection with it; however, it provides only basic functionality for writing bytes to the file. |
| `DataOutputStream` | This class which is in the `java.io` package, allows you to write data of any primitive type or `String` objects to a binary file. The `DataOutputStream` class by itself cannot directly access a file, however. It is used in conjunction with a `FileOutputStream` object that has a connection to a file. |

You wrap a `DataOutputStream` object around a `FileOutputStream` object to write data to a binary file. The following code shows how a file named *MyInfo.dat* can be opened for binary output:

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat");
DataOutputStream outputFile = new DataOutputStream(fstream);
```

The first line creates an instance of the `FileOutputStream` class, which has the ability to open a file for binary output and establish a connection with it. You pass the name of the file that you wish open, as a string, to the constructor. The second line creates an instance of the `DataOutputStream` object that is connected to the `FileOutputStream` referenced by `fstream`. The result of this statement is that the `outputFile` variable will reference an object that is able to write binary data to the *MyInfo.dat* file.

**WARNING!** If the file that you are opening with the `FileOutputStream` object already exists, it will be erased and an empty file by the same name will be created.

**NOTE:** The `FileOutputStream` constructor throws an `IOException` if an error occurs when it attempts to open the file.

If there is no reason to reference the `FileOutputStream` object, then these statements can be combined into one, as follows:

```
DataOutputStream outputFile =
     new DataOutputStream(new FileOutputStream("MyInfo.dat"));
```

Once the `DataOutputStream` object has been created, you can use it to write binary data to the file. Table 11-1 lists some of the `DataOutputStream` methods. Note that each of the methods listed in the table throws an `IOException` if an error occurs.

**Table 11-1** Some of the `DataOutputStream` methods

| Method | Description |
|---|---|
| `void close()` | Closes the file. |
| `void writeBoolean(boolean b)` | Writes the `boolean` value passed to *`b`* to the file. |
| `void writeByte(byte b)` | Writes the `byte` value passed to *`b`* to the file. |
| `void writeChar(int c)` | This method accepts an `int`, which is assumed to be a character code. The character it represents is written to the file as a two-byte Unicode character. |
| `void writeDouble(double d)` | Writes the `double` value passed to *`d`* to the file. |
| `void writeFloat(float f)` | Writes the `float` value passed to *`f`* to the file. |
| `void writeInt(int i)` | Writes the `int` value passed to *`i`* to the file. |
| `void writeLong(long num)` | Writes the `long` value passed to *`num`* to the file. |
| `void writeShort(short s)` | Writes the `short` value passed to *`s`* to the file. |
| `void writeUTF(String str)` | Writes the `String` object passed to *`str`* to the file using the Unicode Text Format. |

The program in Code Listing 11-13 shows a simple demonstration. An array of `int` values is written to the file *Numbers.dat*.

**Code Listing 11-13** (`WriteBinaryFile.java`)

```java
import java.io.*;

/**
   This program opens a binary file and writes the contents
   of an int array to the file.
*/

public class WriteBinaryFile
{
   public static void main(String[] args)
                      throws IOException
   {
      // An array to write to the file
      int[] numbers = { 2, 4, 6, 8, 10, 12, 14 };

      // Create the binary output objects.
      FileOutputStream fstream =
                 new FileOutputStream("Numbers.dat");
      DataOutputStream outputFile =
                 new DataOutputStream(fstream);

      System.out.println("Writing the numbers to the file...");

      // Write the array elements to the file.
      for (int i = 0; i < numbers.length; i++)
         outputFile.writeInt(numbers[i]);

      System.out.println("Done.");

      // Close the file.
      outputFile.close();
   }
}
```

**Program Output**

```
Writing the numbers to the file...
Done.
```

### Reading Data from a Binary File

To open a binary file for input, you use the following classes:

| | |
|---|---|
| `FileInputStream` | This class, which is in the `java.io` package, allows you to open a file for reading binary data and establish a connection with it. It provides only the basic functionality for reading bytes from the file, however. |
| `DataInputStream` | This class, which is in the `java.io` package, allows you to read data of any primitive type, or `String` objects, from a binary file. The `DataInputStream` class by itself cannot directly access a file, however. It is used in conjunction with a `FileInputStream` object that has a connection to a file. |

To open a binary file for input, you wrap a `DataInputStream` object around a `FileInputStream` object. The following code shows the file *MyInfo.dat* can be opened for binary input:

```
FileInputStream fstream = new FileInputStream("MyInfo.dat");
DataInputStream inputFile = new DataInputStream(fstream);
```

The following code, which combines these two statements into one, can also be used:

```
DataInputStream inputFile =
      new DataInputStream(new FileInputStream("MyInfo.dat"));
```

The `FileInputStream` constructor will throw a `FileNotFoundException` if the file named by the string argument cannot be found. Once the `DataInputStream` object has been created, you can use it to read binary data from the file. Table 11-2 lists some of the `DataInputStream` methods. Note that each of the methods listed in the table throws an `EOFException` if the end of the file has already been reached.

**Table 11-2** Some of the `DataInputStream` methods

| Method | Description |
|---|---|
| `void close()` | Closes the file. |
| `boolean readBoolean()` | Reads a `boolean` value from the file and returns it. |
| `byte readByte()` | Reads a `byte` value from the file and returns it. |
| `char readChar()` | Reads a `char` value from the file and returns it. The character is expected to be stored as a two-byte Unicode character, as written by the `DataOutputStream` class's `writeChar` method. |
| `double readDouble()` | Reads a `double` value from the file and returns it. |
| `float readFloat()` | Reads a `float` value from the file and returns it. |
| `int readInt()` | Reads an `int` value from the file and returns it. |
| `long readLong()` | Reads a `long` value from the file and returns it. |
| `short readShort()` | Reads a `short` value from the file and returns it. |
| `String readUTF()` | Reads a string from the file and returns it as a `String` object. The string must have been written with the `DataOutputStream` class's `writeUTF` method. |

The program in Code Listing 11-14 opens the `Numbers.dat` file that was created by the program in Code Listing 11-13. The numbers are read from the file and displayed on the screen. Notice that the program must catch the `EOFException` in order to determine when the file's end has been reached.

**Code Listing 11-14** (ReadBinaryFile.java)

```java
import java.io.*;

/**
   This program opens a binary file, reads
   and displays the contents.
*/

public class ReadBinaryFile
{
   public static void main(String[] args)
                 throws IOException
   {
      int number;                      // A number read from the file
      boolean endOfFile = false;       // EOF flag

      // Create the binary file input objects.
      FileInputStream fstream =
                 new FileInputStream("Numbers.dat");
      DataInputStream inputFile =
                 new DataInputStream(fstream);

      System.out.println("Reading numbers from the file:");

      // Read the contents of the file.
      while (!endOfFile)
      {
         try
         {
            number = inputFile.readInt();
            System.out.print(number + " ");
         }
         catch (EOFException e)
         {
            endOfFile = true;
         }
      }

      System.out.println("\nDone.");

      // Close the file.
      inputFile.close();
   }
}
```

**Program Output**

```
Reading numbers from the file:
2 4 6 8 10 12 14
Done.
```

### Writing and Reading Strings

To write a string to a binary file you should use the `DataOutputStream` class's `writeUTF` method. This method writes its `String` argument in a format known as *UTF-8 encoding*. Here's how the encoding works: Just before writing the string, this method writes a two-byte integer indicating the number of bytes that the string occupies. Then it writes the string's characters in Unicode. (UTF stands for Unicode Text Format.)

When the `DataInputStream` class's `readUTF` method reads from the file, it expects the first two bytes to contain the number of bytes that the string occupies. Then it reads that many bytes and returns them as a `String`.

For example, assuming that `outputFile` references a `DataOutputStream` object, the following code uses the `writeUTF` method to write a string:

```
String name = "Chloe";
outputFile.writeUTF(name);
```

Assuming that `inputFile` references a `DataInputStream` object, the following statement uses the `readUTF` method to read a UTF-8 encoded string from the file:

```
String name = inputFile.readUTF();
```

Remember that the `readUTF` method will correctly read a string only when the string is written with the `writeUTF` method.

This chapter's source code folder contains the example programs *WriteUTF.java* and *ReadUTF.java*, which demonstrate writing and reading strings using these methods.

### Appending Data to an Existing Binary File

If you pass the name of an existing file to the `FileOutputStream` constructor, it will be erased and a new empty file with the same name will be created. Sometimes, however, you want to preserve an existing file and append new data to its current contents. The `FileOutputStream` constructor takes an optional second argument, which must be a `boolean` value. If the argument is `true`, the file will not be erased if it already exists and new data will be written to the end of the file. If the argument is `false`, the file will be erased if it already exists. For example, the following code opens the file *MyInfo.dat* for output. If the file exists, it will not be deleted, and any data written to the file will be appended to the existing data.

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat", true);
DataOutputStream outputFile = new DataOutputStream(fstream);
```

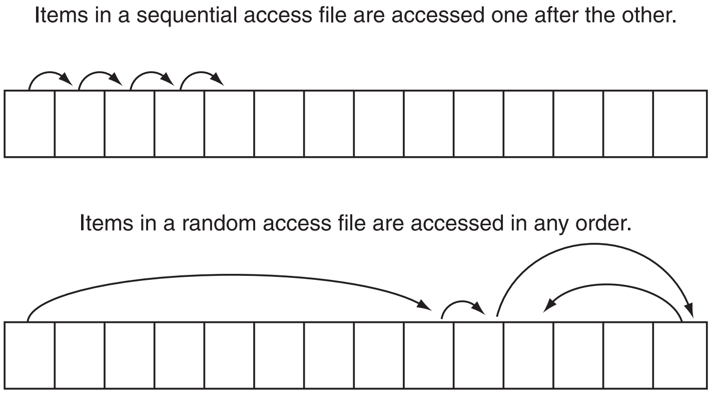
## Random Access Files

All of the programs that you have created to access files so far have performed *sequential file access*. With sequential access, when a file is opened for input, its read position is at the very beginning of the file. This means that the first time data is read from the file, the data will be read from its beginning. As the reading continues, the file's read position advances sequentially through the file's contents.

The problem with sequential file access is that in order to read a specific byte from the file, all the bytes that precede it must be read first. For instance, if a program needs data stored at the hundredth byte of a file, it will have to read the first 99 bytes to reach it. If you've ever listened to a cassette tape player, you understand sequential access. To listen to a song at the end of the tape, you have to listen to all the songs that are before it, or fast-forward over them. There is no way to jump immediately to that particular song.

Although sequential file access is useful in many circumstances, it can slow down a program tremendously. If the file is very large, locating data buried deep inside it can take a long time. Alternatively, Java allows a program to perform *random file access*. In random file access, a program may immediately jump to any location in the file without first reading the preceding bytes. The difference between sequential and random file access is like the difference between a cassette tape and a compact disc. When listening to a CD, there is no need to listen to or fast-forward over unwanted songs. You simply jump to the track that you want to listen to. This is illustrated in Figure 11-15.

**Figure 11-15** Sequential access versus random access


Items in a sequential access file are accessed one after the other.
Items in a random access file are accessed in any order.

To create and work with random access files in Java, you use the `RandomAccessFile` class, which is in the `java.io` package. The general format of the class constructor is as follows:

```
RandomAccessFile(String filename, String mode)
```

The first argument is the name of the file. The second argument is a string indicating the mode in which you wish to use the file. The two modes are `"r"` for reading, and `"rw"` for reading and writing. When a file is opened with `"r"` as the mode, the program can only read from the file. When a file is opened with `"rw"` as the mode, the program can read from the file and write to it. Here are some examples of statements that open files using the `RandomAccessFile` class:

```
// Open a file for random reading.
RandomAccessFile randomFile =
              new RandomAccessFile("MyData.dat", "r");
// Open a file for random reading and writing.
RandomAccessFile randomFile =
              new RandomAccessFile("MyData.dat", "rw");
```

Here are some important points to remember about the two modes:

- If you open a file in `"r"` mode and the file does not exist, a `FileNotFoundException` will be thrown.
- If you open a file in `"r"` mode and try to write to it, an `IOException` will be thrown.
- If you open an existing file in `"rw"` mode, it will not be deleted. The file's existing contents will be preserved.
- If you open a file in `"rw"` mode and the file does not exist, it will be created.

### Reading and Writing with the `RandomAccessFile` Class

A file that is opened or created with the `RandomAccessFile` class is treated as a binary file. In fact, the `RandomAccessFile` class has the same methods as the `DataOutputStream` class for writing data, and the same methods as the `DataInputStream` class for reading data. In fact, you can use the `RandomAccessFile` class to process a binary file sequentially. For example, the program in Code Listing 11-15 opens a file named *Letters.dat* and writes all of the letters of the alphabet to the file.

**Code Listing 11-15** (`WriteLetters.java`)

```
import java.io.*;

/**
   This program uses a RandomAccessFile object to
   create the file Letters.dat. The letters of the
   alphabet are written to the file.
*/

public class WriteLetters
{
   public static void main(String[] args)
                         throws IOException
   {
     // The letters array has all 26 letters.
     char[] letters = {
                         'a', 'b', 'c', 'd', 'e', 'f', 'g',
                         'h', 'i', 'j', 'k', 'l', 'm', 'n',
                         'o', 'p', 'q', 'r', 's', 't', 'u',
                         'v', 'w', 'x', 'y', 'z' };

     System.out.println("Opening the file.");

     // Open a file for reading and writing.
```

```
      RandomAccessFile randomFile =
               new RandomAccessFile("Letters.dat", "rw");

      System.out.println("Writing data to the file...");

      // Sequentially write the letters array to the file.
      for (int i = 0; i < letters.length; i++)
         randomFile.writeChar(letters[i]);

      // Close the file.
      randomFile.close();

      System.out.println("Done.");
      }
   }
```

**Program Output**

```
Opening the file.
Writing data to the file...
Done.
```

After this program executes, the letters of the alphabet will be stored in the *Letters.dat* file. Because the `writeChar` method was used, the letters will each be stored as two-byte characters. This is important to know later when we want to read the characters from the file.

### The File Pointer

The `RandomAccessFile` class treats a file as a stream of bytes. The bytes are numbered, with the first byte being byte 0. The last byte's number is one less than the number of bytes in the file. These byte numbers are similar to an array's subscripts, and are used to identify locations in the file.

Internally, the `RandomAccessFile` class keeps a long integer value known as the file pointer. The *file pointer* holds the byte number of a location in the file. When a file is first opened, the file pointer is set to 0. This causes it to "point" to the first byte in the file. When an item is read from the file, it is read from the byte that the file pointer points to. Reading also causes the file pointer to advance to the byte just beyond the item that was read. For example, let's say the file pointer points to byte 0 and an `int` is read from the file with the `readInt` method. An `int` is four bytes in size, so four bytes will be read from the file, starting at byte 0. After the value is read, the file pointer will be advanced to byte number 4, which is the 5th byte in the file. If another item is immediately read, the reading will begin at byte number 4. If the file pointer refers to a byte number that is beyond the end of the file, an `EOFException` is thrown when a read operation is performed.

Writing also takes place at the location pointed to by the file pointer. If the file pointer points to the end of the file when a write operation is performed, then the data will be written to the end of the file. However, if the file pointer holds the number of a byte within the file, at a location where data is already stored, then a write operation will cause data to be written over the existing data at that location.

Not only does the `RandomAccessFile` class let you read and write data, but also it allows you to move the file pointer. This means that you can immediately read data from any byte location in the file. It also means that you can write data to any location in the file, over existing data. To move the file pointer, you use the `seek` method. Here is the method's general format:

```
void seek(long position)
```

The argument is the number of the byte that you want to move the file pointer to. For example, look at the following code:

```
RandomAccessFile file =
            new RandomAccessFile("MyInfo.dat", "r");
file.seek(99);
byte b = file.readByte();
```

This code opens the file *MyInfo.dat* for reading. The `seek` method is called to move the file pointer to byte number 99 (which is the 100th byte in the file). Then, the `readByte` method is called to read byte number 99 from the file. After that statement executes, the file pointer will be advanced by one byte, so it will point to byte 100. Suppose we continue processing the same file with the following code:

```
file.seek(49);
int i = file.readInt();
```

First, the `seek` method moves the file pointer to byte number 49 (which is the 50th byte in the file). Then, the `readInt` method is called. This reads an `int` from the file. An `int` is four bytes in size, so this statement reads four bytes, beginning at byte number 49. After the statement executes the file pointer will be advanced by four bytes, so it will point to byte 53.

Although a file might contain `char`s, `int`s, `double`s, strings, and so forth, the `RandomAccessFile` class sees it only as a stream of bytes. The class is unaware of the data types of the data stored in the file, and it cannot determine where one item of data ends and another begins. When you write a program that reads data from a random access file, it is your responsibility to know how the data is structured.

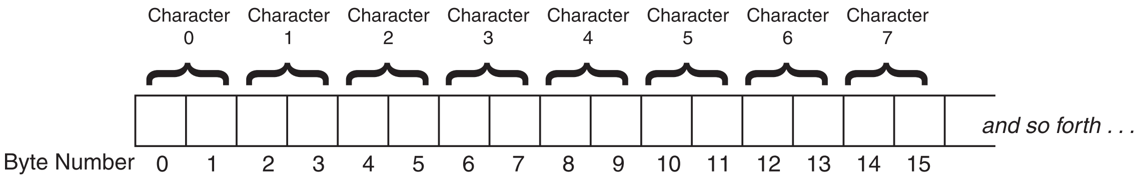
For example, recall that the program in Code Listing 11-15 wrote the letters of the alphabet to the *Letters.dat* file. Let's say the first letter is character `0`, the second letter is character `1`, and so forth. Suppose we want to read character `5` (the sixth letter in the file). At first, we might be tempted to try the following code:

```
// Open the file for reading.
RandomAccessFile randomFile =
              new RandomAccessFile("Letters.dat", "r");
// Move the file pointer to byte 5, which is the 6th byte.
randomFile.seek(5);
// Read the character.
char ch = randomFile.readChar();
// What will this display?
System.out.println("The sixth letter is " + ch);
```

Although this code will compile and run, you might be surprised at the result. Recall that the `writeChar` method writes a character as two bytes. Because each character occupies two bytes in the file, the sixth character begins at byte 10, not byte 5. This is illustrated in

Figure 11-16. In fact, if we try to read a character starting at byte 5, we will read garbage because byte 5 is not at the beginning of a character.

**Figure 11-16** Layout of the *Letters.dat* file



To determine the position of a character in the file, we must take each character's size into account. The following code will correctly read and display the sixth character. To determine the character's starting byte number, it multiplies the size of a character by the number of the character we want to locate.

```
final int CHAR_SIZE = 2;  // Each char uses two bytes
// Move the file pointer to character 5.
randomFile.seek(CHAR_SIZE * 5);
// Read the character.
char ch = randomFile.readChar();
// This will display the correct character.
System.out.println("The sixth character is " + ch);
```

The program in Code Listing 11-16 demonstrates further. It randomly reads characters 5, 10, and 3 from the file.

**Code Listing 11-16** (ReadRandomLetters.java)

```
import java.io.*;

/**
   This program uses the RandomAccessFile class to open
   the file Letters.dat and randomly read letters from
   different locations.
*/

public class ReadRandomLetters
{
   public static void main(String[] args) throws IOException
   {
      final int CHAR_SIZE = 2;    // 2 byte characters
      long byteNum;               // The byte number
      char ch;                    // A character from the file

      // Open the file for reading.
      RandomAccessFile randomFile =
                   new RandomAccessFile("Letters.dat", "r");

      // Move to the character 5. This is the 6th
      // character from the beginning of the file.
      byteNum = CHAR_SIZE * 5;
      randomFile.seek(byteNum);

      // Read the character stored at this location
      // and display it. Should be the letter f.
      ch = randomFile.readChar();
      System.out.println(ch);

      // Move to character 10 (the 11th character),
      // read the character, and display it.
      // Should be the letter k.
      byteNum = CHAR_SIZE * 10;
      randomFile.seek(byteNum);
      ch = randomFile.readChar();
      System.out.println(ch);

      // Move to character 3 (the 4th character),
      // read the character, and display it.
      // Should be the letter d.
      byteNum = CHAR_SIZE * 3;
      randomFile.seek(byteNum);
      ch = randomFile.readChar();
      System.out.println(ch);

      // Close the file.
      randomFile.close();
   }
}
```

**Program Output**

```
f
k
d
```

See Appendix H —Working with Records and Random Access Files. The appendix is available on the book’s companion Web site, at www.pearsonhighered.com.gaddis.

## Object Serialization

In Appendix H, available on the book’s companion Web site, at www.pearsonhighered.com.gaddis, you can see how an object’s fields can be retrieved and saved to a file as fields in a record. If an object contains other types of objects as fields, however, the process of saving its contents can become complicated. Fortunately, Java allows you to *serialize* objects, which is a simpler way of saving objects to a file.

When an object is serialized, it is converted into a series of bytes that contain the object's data. If the object is set up properly, even the other objects that it might contain as fields are automatically serialized. The resulting set of bytes can be saved to a file for later retrieval.

In order for an object to be serialized, its class must implement the `Serializable` interface. The `Serializable` interface, which is in the `java.io` package, has no methods or fields. It is used only to let the Java compiler know that objects of the class might be serialized. In addition, if a class contains objects of other classes as fields, those classes must also implement the `Serializable` interface, in order to be serialized.

For example, in this chapter's source code folder there is a modified version of the `BankAccount` class named `BankAccount2`. The only modification to the class is that it implements the `Serializable` interface. Here are the modified lines of code from the file:

```
import java.io.Serializable;

public class BankAccount2 implements Serializable
```

This new code tells the compiler that we want to be able to serialize objects of the `BankAccount2` class. To write a serialized object to a file, you use an `ObjectOutputStream` object. The `ObjectOutputStream` class is designed to perform the serialization process (converting an object to a series of bytes). To write the bytes to a file, you must also use an output stream object, such as `FileOutputStream`. Here is an example:

```
FileOutputStream outStream =
             new FileOutputStream("Objects.dat");
ObjectOutputStream objectOutputFile =
             new ObjectOutputStream(outStream);
```

To serialize an object and write it to the file, use the `ObjectOutputStream` class's `writeObject` method, as shown here:

```
BankAccount2 account = new BankAccount2(5000.0);
objectOutputFile.writeObject(account);
```

The `writeObject` method throws an `IOException` if an error occurs.

The process of reading a serialized object's bytes and constructing an object from them is known as *deserialization*. To deserialize an object you use an `ObjectInputStream` object, along with a `FileInputStream` object. Here is an example of how to set up the objects:

```
FileInputStream inStream =
             new FileInputStream("Objects.dat");
ObjectInputStream objectInputFile =
             new ObjectInputStream(inStream);
```

To read a serialized object from the file, use the `ObjectInputStream` class's `readObject` method. Here is an example:

```
BankAccount2 account;
account = (BankAccount2) objectInputFile.readObject();
```

The `readObject` method returns the deserialized object. Notice that you must cast the return value to the desired class type. (The `readObject` method throws a number of different exceptions if an error occurs. See the API documentation for more information.)

The following programs demonstrate how to serialize and deserialize objects. The program in Code Listing 11-17 serializes three `BankAccount2` objects, and the program in Code Listing 11-18 deserializes them.

**Code Listing 11-17** (`SerializeObjects.java`)

```java
import java.io.*;
import java.util.Scanner;

/**
   This program serializes the objects in an array of
   BankAccount2 objects.
*/

public class SerializeObjects
{
   public static void main(String[] args)
                     throws IOException
   {
      double balance;             // An account balance
      final int NUM_ITEMS = 3;    // Number of accounts

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create a BankAccount2 array
      BankAccount2[] accounts =
                    new BankAccount2[NUM_ITEMS];

      // Populate the array.
      for (int i = 0; i < accounts.length; i++)
      {
         // Get an account balance.
         System.out.print("Enter the balance for " +
                          "account " + (i + 1) + ": ");
         balance = keyboard.nextDouble();

         // Create an object in the array.
         accounts[i] = new BankAccount2(balance);
      }

      // Create the stream objects.
      FileOutputStream outStream =
                 new FileOutputStream("Objects.dat");
      ObjectOutputStream objectOutputFile =
                 new ObjectOutputStream(outStream);

      // Write the serialized objects to the file.
      for (int i = 0; i < accounts.length; i++)
      {
         objectOutputFile.writeObject(accounts[i]);
      }

      // Close the file.
      objectOutputFile.close();

      System.out.println("The serialized objects " +
             "were written to the Objects.dat file.");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter the balance for account 1: 5000.0 [Enter]
Enter the balance for account 2: 2500.0 [Enter]
Enter the balance for account 3: 1800.0 [Enter]
The serialized objects were written to the Objects.dat file.
```

**Code Listing 11-18** (`DeserializeObjects.java`)

```
import java.io.*;

/**
   This program deserializes the objects in the Objects.dat
   file and stores them in an array.
*/

public class DeserializeObjects
{
   public static void main(String[] args)
                       throws Exception
   {
      double balance;              // An account balance
      final int NUM_ITEMS = 3;     // Number of accounts

      // Create the stream objects.
      FileInputStream inStream =
              new FileInputStream("Objects.dat");
      ObjectInputStream objectInputFile =
              new ObjectInputStream(inStream);

      // Create a BankAccount2 array
      BankAccount2[] accounts =
              new BankAccount2[NUM_ITEMS];

      // Read the serialized objects from the file.
      for (int i = 0; i < accounts.length; i++)
```

```
         {
            accounts[i] =
              (BankAccount2) objectInputFile.readObject();
         }

         // Close the file.
         objectInputFile.close();

         // Display the objects.
         for (int i = 0; i < accounts.length; i++)
         {
             System.out.println("Account " + (i + 1) +
                      " $ " + accounts[i].getBalance());
         }
      }
   }
```

**Program Output**

```
Account 1 $ 5000.0
Account 2 $ 2500.0
Account 3 $ 1800.0
```

## Serializing Aggregate Objects

If a class implements the `Serializable` interface, then all of the fields in that class must be serializable. This isn't a problem for primitive variables because they are serializable just as they are. However, if the class has a reference variable as a field, then the object referenced by that variable should also be serializable. This means that the object's class should also implement the `Serializable` interface. If it doesn't, then the `transient` key word should be used in the reference variable's declaration. Here is an example:

```
private transient SomeClass refVar;
```

Because of the `transient` key word, the compiler will skip the object referenced by `refVar` during the serialization process. Fortunately, the `String` class, and most of the other classes found in the Java API, implement the `Serializable` interface.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

11.20 What is the difference between a text file and a binary file?

11.21 What classes do you use to write output to a binary file? What classes do you use to read from a binary file?

11.22 What is the difference between sequential and random access?

11.23 What class do you use to work with random access files?

11.24 What are the two modes that a random access file may be opened in? Explain the difference between them.

11.25 What must you do to a class in order to serialize objects of that class?

## 11.4 Common Errors to Avoid

- **Assuming that all statements inside a try block will execute.** When an exception is thrown, the try block is exited immediately. This means that statements appearing in the try block after the offending statement will not be executed.
- **Getting the `try`, `catch`, and `finally` clauses out of order.** In a `try` statement, the `try` clause must appear first, followed by all of the `catch` clauses, followed by the optional `finally` clause.
- **Writing two `catch` clauses that handle the same exception in the same `try` statement.** You cannot have more than one `catch` clause per exception type in the same `try` statement.
- **When catching multiple exceptions that are related to one another through inheritance, listing the more general exceptions first.** If you are handling multiple exceptions in the same `try` statement, and some of the exceptions are related to each other through inheritance, then you should handle the more specialized exception classes before the more general exception classes. Otherwise, an error will occur because the compiler thinks that you are handling the same exception more than once.
- **Forgetting to write a `throws` clause on a method that can throw a checked exception but does not handle the exception.** If a method is capable of throwing a checked exception but does not handle the exception, it must have a `throws` clause in its header that specifies the exception.
- **Calling a method but not handling an exception that it might throw.** You must either handle all of the checked exceptions that a method can throw, or list them in the calling method's `throws` clause.
- **In a custom exception class, forgetting to pass an error message to the superclass's constructor.** If you do not pass an error message to the superclass's constructor, the exception object will have a `null` error message.
- **Serializing an object with members that are not serializable.** If a class has fields that are objects of other classes, those classes must implement the `Serializable` interface in order to be serialized.

## Review Questions and Exercises

### Multiple Choice and True/False

1. When an exception is generated, it is said to have been __________.
   a. built
   b. thrown
   c. caught
   d. killed
2. This is a section of code that gracefully responds to exceptions.
   a. exception generator
   b. exception manipulator
   c. exception handler
   d. exception monitor

3. If your code does not handle an exception when it is thrown, it is dealt with by this.
   a. default exception handler
   b. the operating system
   c. system debugger
   d. default exception generator
4. All exception classes inherit from this class.
   a. `Error`
   b. `RuntimeException`
   c. `JavaException`
   d. `Throwable`
5. `FileNotFoundException` inherits from __________.
   a. `Error`
   b. `IOException`
   c. `JavaException`
   d. `FileException`
6. You can think of this code as being “protected” because the application will not halt if it throws an exception.
   a. try block
   b. catch block
   c. finally block
   d. protected block
7. This method can be used to retrieve the error message from an exception object.
   a. `errorMessage`
   b. `errorString`
   c. `getError`
   d. `getMessage`
8. The numeric wrapper classes’ “parse” methods all throw an exception of this type.
   a. `ParseException`
   b. `NumberFormatException`
   c. `IOException`
   d. `BadNumberException`
9. This is one or more statements that are always executed after the try block has executed and after any catch blocks have executed if an exception was thrown.
   a. try block
   b. catch block
   c. finally block
   d. protected block
10. This is an internal list of all the methods that are currently executing.
   a. invocation list
   b. call stack
   c. call list
   d. list trace

11. This method may be called from any exception object, and it shows the chain of methods that were called when the exception was thrown.
    a. `printInvocationList`
    b. `printCallStack`
    c. `printStackTrace`
    d. `printCallList`
12. These are exceptions that inherit from the `Error` class or the `RuntimeException` class.
    a. unrecoverable exceptions
    b. unchecked exceptions
    c. recoverable exceptions
    d. checked exceptions
13. All exceptions that do *not* inherit from the `Error` class or the `RuntimeException` class are __________.
    a. unrecoverable exceptions
    b. unchecked exceptions
    c. recoverable exceptions
    d. checked exceptions
14. This informs the compiler of the exceptions that could get thrown from a method.
    a. `throws` clause
    b. parameter list
    c. `catch` clause
    d. method return type
15. You use this statement to throw an exception manually.
    a. `try`
    b. `generate`
    c. `throw`
    d. `System.exit(0)`
16. This is the process of converting an object to a series of bytes that represent the object's data.
    a. serialization
    b. deserialization
    c. dynamic conversion
    d. casting
17. **True or False:** You are not required to catch exceptions that inherit from the `RuntimeException` class.
18. **True or False:** When an exception is thrown by code inside a try block, all of the statements in the try block are always executed.
19. **True or False:** `IOException` serves as a superclass for exceptions that are related to programming errors, such as an out-of-bounds array subscript.
20. **True or False:** You cannot have more than one `catch` clause per `try` statement.

21. **True or False:** When an exception is thrown, the JVM searches the `try` statement's `catch` clauses from top to bottom and passes control of the program to the first `catch` clause with a parameter that is compatible with the exception.
22. **True or False:** Not including polymorphic references, a `try` statement may have only one `catch` clause for each specific type of exception.
23. **True or False:** When in the same `try` statement you are handling multiple exceptions and some of the exceptions are related to each other through inheritance, you should handle the more general exception classes before the more specialized exception classes.
24. **True or False:** The `throws` clause causes an exception to be thrown.

## Find the Error

Find the error in each of the following code segments:

1.
```
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
try
{
   File file = new File("MyFile.txt");
   Scanner inputFile = new Scanner(file);
}
```

2.
```
// Assume inputFile references a Scanner object.
try
{
   input = inputFile.nextInt();
}
finally
{
   inputFile.close();
}
catch (InputMismatchException e)
{
   System.out.println(e.getMessage());
}
```

3.
```
try
{
   number = Integer.parseInt(str);
}
catch (Exception e)
{
   System.out.println(e.getMessage());
}
```

```
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
```

## Algorithm Workbench

1. Look at the following program and tell what it will output when run:

```
public class ExceptionTest

{
   public static void main(String[] args)
   {

      int number;
      String str;

      try
      {
         str = "xyz";
         number = Integer.parseInt(str);
         System.out.println("A");
      }
      catch(NumberFormatException e)
      {
         System.out.println("B");
      }
      catch(IllegalArgumentException e)
      {
         System.out.println("C");
      }
      System.out.println("D");
   }
}
```

2. Look at the following program and tell what it will output when run:

```
public class ExceptionTest
{
   public static void main(String[] args)
   {
      int number;
      String str;
```

```
         try
         {
            str = "xyz";
            number = Integer.parseInt(str);
            System.out.println("A");
         }
         catch(NumberFormatException e)
         {
            System.out.println("B");
         }
         catch(IllegalArgumentException e)
         {
            System.out.println("C");
         }
         finally
         {
            System.out.println("D");
         }

         System.out.println("E");
      }
   }
```

3. Write a method that searches a numeric array for a specified value. The method should return the subscript of the element containing the value if it is found in the array. If the value is not found, the method should throw an exception of the `Exception` class with the error message "Element not found".
4. Write a statement that throws an `IllegalArgumentException` with the error message "Argument cannot be negative".
5. Write an exception class that can be thrown when a negative number is passed to a method.
6. Write a statement that throws an instance of the exception class that you created in Algorithm Workbench 5.
7. The method `getValueFromFile` is public and returns an `int`. It accepts no arguments. The method is capable of throwing an `IOException` and a `FileNotFoundException`. Write the header for this method.
8. Write a `try` statement that calls the `getValueFromFile` method described in Algorithm Workbench 7. Be sure to handle all the exceptions that the method can throw.
9. Write a statement that creates an object that can be used to write binary data to the file *Configuration.dat*.
10. Write a statement that opens the file *Customers.dat* as a random access file for both reading and writing.
11. Assume that the reference variable `r` refers to a serializable object. Write code that serializes the object to the file *ObjectData.dat*.

### Short Answer

1. What is meant when it is said that an exception is thrown?
2. What does it mean to catch an exception?
3. What happens when an exception is thrown, but the `try` statement does not have a `catch` clause that is capable of catching it?
4. What is the purpose of a `finally` clause?
5. Where does execution resume after an exception has been thrown and caught?
6. When multiple exceptions are caught in the same `try` statement and some of them are related through inheritance, does the order in which they are listed matter?
7. What types of objects can be thrown?
8. When are you required to have a `throws` clause in a method header?
9. What is the difference between a checked exception and an unchecked exception?
10. What is the difference between the `throw` statement and the `throws` clause?
11. What is the difference between a text file and a binary file?
12. What is the difference between a sequential access file and a random access file?
13. What happens when you serialize an object? What happens when you deserialize an object?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. `TestScores` Class

Write a class named `TestScores`. The class constructor should accept an array of test scores as its argument. The class should have a method that returns the average of the test scores. If any test score in the array is negative or greater than 100, the class should throw an `IllegalArgumentException`. Demonstrate the class in a program.

### 2. `TestScores` Class Custom Exception

Write an exception class named `InvalidTestScore`. Modify the `TestScores` class you wrote in Programming Challenge 1 so that it throws an `InvalidTestScore` exception if any of the test scores in the array are invalid.

### 3. `RetailItem` Exceptions

Programming Challenge 4 of Chapter 6 required you to write a `RetailItem` class that holds data pertaining to a retail item. Write an exception class that can be instantiated and thrown when a negative number is given for the price. Write another exception class that can be instantiated and thrown when a negative number is given for the units on hand. Demonstrate the exception classes in a program.

### 4. `Month` Class Exceptions

Programming Challenge 5 of Chapter 8 required you to write a `Month` class that holds information about the month. Write exception classes for the following error conditions:

- A number less than 1 or greater than 12 is given for the month number.
- An invalid string is given for the name of the month.

Modify the `Month` class so that it throws the appropriate exception when either of these errors occurs. Demonstrate the classes in a program.

### 5. `Payroll` Class Exceptions

Programming Challenge 5 of Chapter 6 required you to write a `Payroll` class that calculates an employee's payroll. Write exception classes for the following error conditions:

- An empty string is given for the employee's name.
- An invalid value is given for the employee's ID number. If you implemented this field as a string, then an empty string would be invalid. If you implemented this field as a numeric variable, then a negative number or zero would be invalid.
- An invalid number is given for the number of hours worked. This would be a negative number or a number greater than 84.
- An invalid number is given for the hourly pay rate. This would be a negative number or a number greater than 25.

Modify the `Payroll` class so that it throws the appropriate exception when any of these errors occurs. Demonstrate the exception classes in a program.

### 6. `FileArray` Class

Design a class that has a static method named `writeArray`. The method should take two arguments: the name of a file and a reference to an `int` array. The file should be opened as a binary file, the contents of the array should be written to the file, and then the file should be closed.

Write a second method in the class named `readArray`. The method should take two arguments: the name of a file and a reference to an `int` array. The file should be opened, data should be read from the file and stored in the array, and then the file should be closed. Demonstrate both methods in a program.

### 7. File Encryption Filter

File encryption is the science of writing the contents of a file in a secret code. Your encryption program should work like a filter, reading the contents of one file, modifying the data into a code, and then writing the coded contents out to a second file. The second file will be a version of the first file, but written in a secret code.

Although there are complex encryption techniques, you should come up with a simple one of your own. For example, you could read the first file one character at a time, and add 10 to the character code of each character before it is written to the second file.

### 8. File Decryption Filter

Write a program that decrypts the file produced by the program in Programming Challenge 7. The decryption program should read the contents of the coded file, restore the data to its original state, and write it to another file.

### 9. `TestScores` Modification for Serialization

Modify the `TestScores` class that you created for Programming Challenge 1 to be serializable. Write a program that creates an array of at least five `TestScore` objects and serializes them. Write another program that deserializes the objects from the file.

### 10. Exception Project

VideoNote

The Exception Project Problem

This assignment assumes you have completed Programming Challenge 1 of Chapter 10 (`Employee` and `ProductionWorker` Classes). Modify the `Employee` and `ProductionWorker` classes so they throw exceptions when the following errors occur:

- The `Employee` class should throw an exception named `InvalidEmployeeNumber` when it receives an invalid employee number.
- The `ProductionWorker` class should throw an exception named `InvalidShift` when it receives an invalid shift.
- The `ProductionWorker` class should throw an exception named `InvalidPayRate` when it receives a negative number for the hourly pay rate.

Write a test program that demonstrates how each of these exception conditions works.

# CHAPTER 12 JavaFX: GUI Programming and Basic Controls

**TOPICS**

12.1 Graphical User Interfaces
12.2 Introduction to JavaFX
12.3 Creating Scenes
12.4 Displaying Images
12.5 More about the `HBox`, `VBox`, and `GridPane` Layout Containers
12.6 `Button` Controls and Events
12.7 Reading Input with `TextField` Controls
12.8 Using Anonymous Inner Classes and Lambda Expressions to Handle Events
12.9 The `BorderPane` Layout Container
12.10 The `ObserveableList` Interface
12.11 Common Errors to Avoid

## 12.1 Graphical User Interfaces

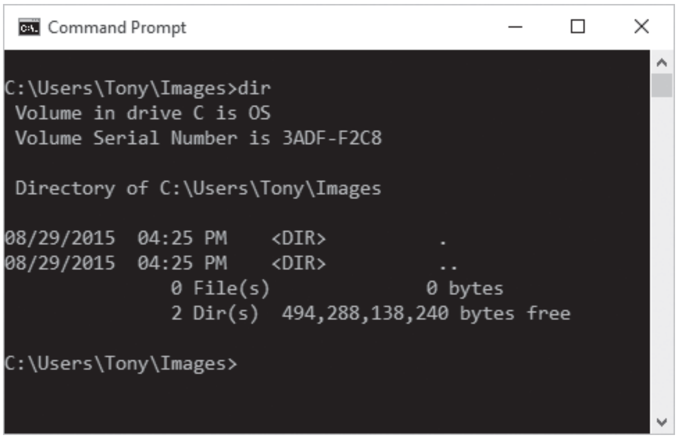
**CONCEPT: A graphical user interface allows the user to interact with a program using graphical elements such as windows, icons, buttons, and so on.**

Programmers commonly employ the term *user* to describe any hypothetical person who might be using a computer and its programs. A computer's *user interface* is the part of the computer with which the user interacts. One part of the user interface consists of hardware devices, such as the keyboard and video display. Another part of the user interface involves the way the computer's operating system and application software accepts commands from the user. For many years, the only way that the user could interact with a computer was through a command line interface. A *command line interface*, which is also known as a *console interface*, requires the user to type commands. If a command is typed correctly, it is executed and the results are displayed. If a command is not typed correctly, an error message is displayed. Figure 12-1 shows the Windows command prompt window, which is an example of a command line interface.

Many computer users, especially beginners, find command line interfaces difficult to use. This is because there are many commands to be learned, and each command has its own syntax, much like a programming statement. If a command isn't entered correctly, it will not work.

In the 1980s, a new type of interface known as a graphical user interface came into use in commercial operating systems. A *graphical user interface*, or *GUI* (pronounced "gooey"), allows the user to interact with the operating system and application programs through

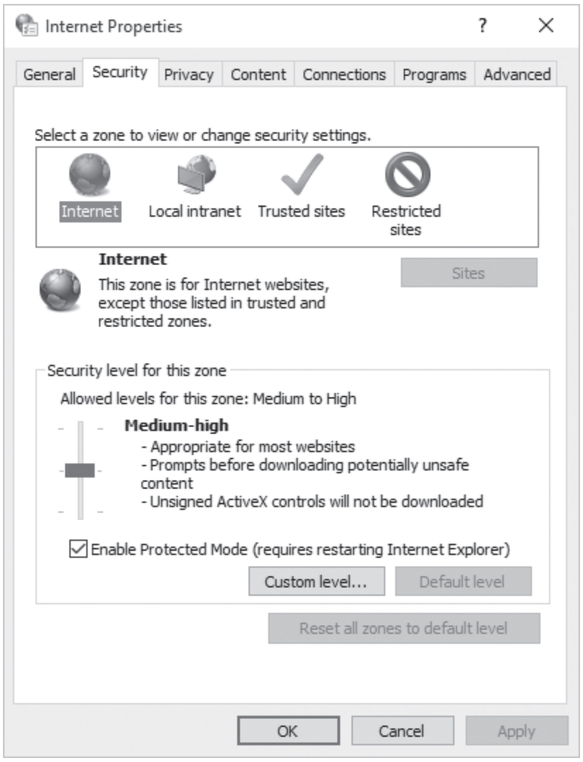
**Figure 12-1** A command line interface (Microsoft Corporation)


Command Prompt

```
C:\Users\Tony\Images>dir
 Volume in drive C is OS
 Volume Serial Number is 3ADF-F2C8

 Directory of C:\Users\Tony\Images

08/29/2015  04:25 PM    <DIR>          .
08/29/2015  04:25 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  494,288,138,240 bytes free

C:\Users\Tony\Images>
```

graphical elements on the screen. GUIs also popularized the use of the mouse as an input device. Instead of requiring the user to type commands on the keyboard, GUIs allow the user to point at graphical elements and click the mouse button to activate them. Much of the interaction with a GUI is done through windows that display information and allow the user to perform actions. Figure 12-2 shows an example of a window that allows the user to change the system's Internet settings. Instead of typing cryptic commands, the user interacts with icons, buttons, and slider bars.

**Figure 12-2** A window in a graphical user interface (Microsoft Corporation)


Internet Properties
?
General
Security
Privacy
Content
Connections
Programs
Advanced
Internet
Local intranet
Trusted sites
Restricted sites
Sites
Custom level...
Default level
Reset all zones to default level
OK
Cancel
Apply

Select a zone to view or change security settings.

**Internet**

This zone is for Internet websites, except those listed in trusted and restricted zones.

Security level for this zone

Allowed levels for this zone: Medium to High

**Medium-high**

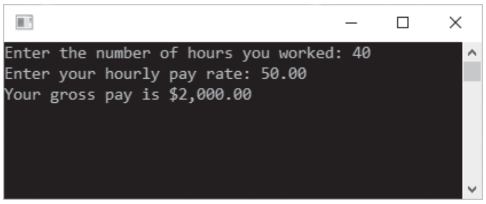
- Appropriate for most websites
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded

☑ Enable Protected Mode (requires restarting Internet Explorer)

## Event-Driven GUI Programs

In a text-based environment, such as a command line interface, programs determine the order in which things happen. For example, Figure 12-3 shows the interaction that has taken place in a text environment with a program that calculates an employee's gross pay. First, the program prompted the user to enter the number of hours worked. In the figure, the user entered 40 and pressed the Enter key. Next, the program prompted the user to enter his or her hourly pay rate. In the figure, the user entered 50.00, and pressed the Enter key. Then, the program displayed the user's gross pay. As the program was running, the user had no choice but to enter the data in the order requested.

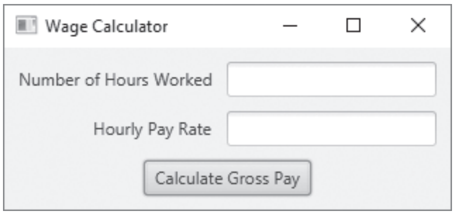
**Figure 12-3** Interaction with a program in a text environment (Oracle Corporate Counsel)



```
Enter the number of hours you worked: 40
Enter your hourly pay rate: 50.00
Your gross pay is $2,000.00
```

In a GUI environment, however, the user determines the order in which things happen. For example, Figure 12-4 shows a GUI application that calculates an employee's gross pay. Notice there are boxes into which the user enters the number of hours worked and the hourly pay rate. The user can enter the hours and the pay rate in any order he or she wishes. If the user makes a mistake, he or she can erase the data that was entered and retype it. When the user is ready to calculate the area, he or she clicks the *Calculate Gross Pay* button, and the program performs the calculation.

**Figure 12-4** A GUI application (Oracle Corporate Counsel)



Because GUI programs must respond to the actions of the user, they are said to be *event-driven*. The user causes events, such as the clicking of a button, and the program responds to those events.

This chapter focuses exclusively on the development of GUI applications using the JavaFX library. As you work through this chapter, you will learn to create applications that interact with the user through windows containing graphical objects. You will also learn how to program your applications to respond to the events that take place as the user interacts with them.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.1 What is a user interface?

12.2 How does a command line interface work?

12.3 When the user runs a program in a text-based environment, such as the command line interface, what determines the order in which things happen?

12.4 What is an event-driven program?

## 12.2 Introduction to JavaFX

**CONCEPT: In Java, you can use the JavaFX library to create GUI and graphical applications. JavaFX is the next generation GUI toolkit for Java developers.**

VideoNote
Introduction to JavaFX

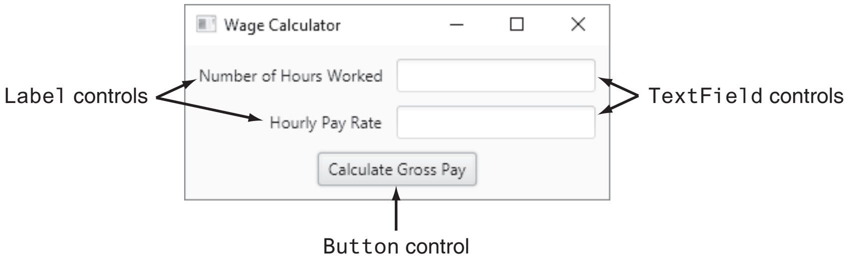
In this chapter, we will use JavaFX to create GUI applications. *JavaFX* is a standard Java library for developing rich applications that employ graphics. You can use it to create GUI applications, as well as applications that display 2D and 3D graphics. You can use JavaFX to create standalone graphics applications that run on your local computer, applications that run from a remote server, or applications that are embedded in a Web page. This chapter introduces you to JavaFX as a tool for creating standalone GUI applications.

### Controls

The objects that are visible in a program's graphical user interface are commonly referred to as *controls*. The GUI shown in Figure 12-5 contains two `Label` controls, two `TextField` controls, and a `Button` control. Here is a summary of each of these types of controls:

| | |
|---|---|
| `Label` control | A `Label` control displays text. The window shown in Figure 12-5 contains two `Label` controls. One of the `Label` controls displays the text *Number of Hours Worked*, and the other `Label` control displays the text *Hourly Pay Rate*. |
| `TextField` control | A `TextField` control appears as a rectangular region that can accept keyboard input from the user. The window shown in Figure 12-5 has two `TextField` controls: one in which the user enters the number of hours worked, and another in which the user enters the hourly pay rate. |
| `Button` control | A `Button` is a rectangular control that appears as a button with a caption written across its face. When the user clicks a `Button` control (either with a mouse or by touching it on a touch screen), an action takes place. The window in Figure 12-5 has one `Button` control, showing the caption *Calculate Gross Pay*. When the user clicks this button, the program calculates and displays the gross pay. |

**Figure 12-5** Controls in a GUI (Oracle Corporate Counsel)



`Labels`, `TextFields`, and `Buttons` are just a few of the controls you will learn to use. As you study this chapter, you will create applications that incorporate many different types of controls.

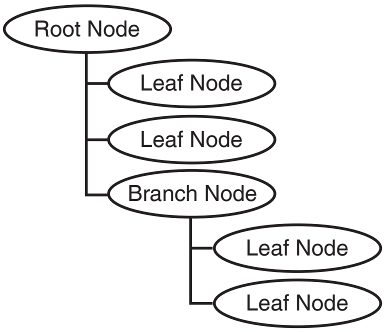
## Stages and Scenes

When programming with JavaFX, you use a theater metaphor to describe the elements of a GUI. Imagine that you are seated in a theater watching a play. There is a stage, and on that stage, a scene is being performed.

A JavaFX application also has a stage and a scene. The stage is a window, and the scene is a collection of GUI objects (`Labels`, `TextFields`, `Buttons`, and other controls) contained within the window. You can think of the GUI objects as the actors that make up the scene.

In memory, the GUI objects in a scene are organized as nodes in a *scene graph*, which is a tree-like hierarchical data structure. Figure 12-6 shows an example.

**Figure 12-6** Nodes in a scene graph



A scene graph can have three types of nodes:

- *Root Node*: There is only one root node, which is the parent of all the other nodes in the scene graph.
- *Branch Node*: A branch node can have other nodes as children.
- *Leaf Node*: A leaf node cannot have children.

In a nutshell, the root node and branch nodes can have children, but leaf nodes cannot. Leaf nodes are typically controls with which the user interacts, such as `Labels`, `TextFields`, and `Buttons`. Branch nodes are objects that serve as containers for other nodes.

## The `Application` Class

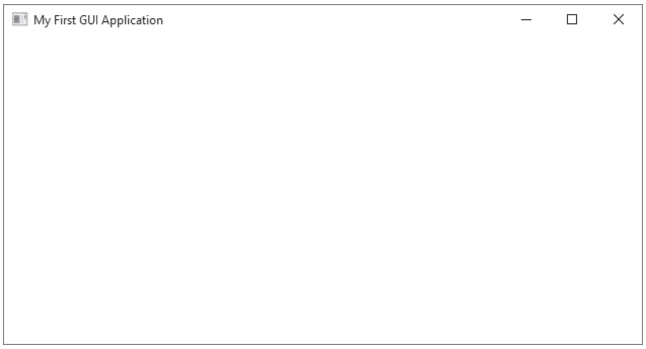
The JavaFX library has an abstract class named `Application` that is the foundation of a GUI application. All JavaFX applications must extend the `Application` class.

The `Application` class has an abstract method named `start`, which is the entry point for the application. Because the `start` method is abstract, you must override it. Let's look at an example. The program in Code Listing 12-1 is a simple JavaFX application that displays an empty window, as shown in Figure 12-7. The program creates a stage, but does not create a scene.

**Code Listing 12-1** (`MyFirstGUI.java`)

```
 1  import javafx.application.Application;
 2  import javafx.stage.Stage;
 3
 4  /**
 5    A simple JavaFX GUI application
 6  */
 7
 8  public class MyFirstGUI extends Application
 9  {
10     public static void main(String[] args)
11     {
12        // Launch the application.
13        launch(args);
14     }
15
16     @Override
17     public void start(Stage primaryStage)
18     {
19        // Set the stage title.
20        primaryStage.setTitle("My First GUI Application");
21
22        // Show the window.
23        primaryStage.show();
24     }
25  }
```

**Figure 12-7** Window displayed by `MyFirstGUI.java` (Oracle Corporate Counsel)



Let's take a closer look at the program in Code Listing 12-1:

- Line 1 imports the `Application` class, which is in the `javafx.application` package.
- Line 2 imports the `Stage` class, which is in the `javafx.stage` package. (You will see this class used in the `start` method.)
- The declaration for the `MyFirstGUI` class begins in line 8. Notice that it extends the `Application` class.

- The `main` method appears in lines 10 through 14. The `main` method does only one thing: it calls the `launch` method in line 13. The `launch` method, which is inherited from the `Application` class, does a number of setup operations, including the following:
    - It creates a `Stage` object that will be the application's window.
    - It calls the `start` method, passing a reference to the `Stage` object as an argument.
- The `start` method appears in lines 17 through 24. This is the entry point for the application. As previously mentioned, the `start` method is an abstract method in the `Application` class, and we must override it. Notice the method has a parameter named `primaryStage`. The `primaryStage` variable will reference the `Stage` object created by the `launch` method.
- Line 20 calls the `Stage` object's `setTitle` method. This sets the text to be displayed in the window's title bar.
- Line 23 calls the `Stage` object's `show` method. This displays the application's window.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.5 What is JavaFX?

12.6 Is a window the same as a stage, or a scene?

12.7 What is a scene graph?

12.8 What are the three types of nodes in a scene graph?

12.9 What type of scene graph node can have children? What type cannot?

12.10 The JavaFX library has an abstract class that is the foundation of a GUI application. What is the name of the class?

12.11 What is the purpose of the `launch` method of the `Application` class?

12.12 What is the purpose of the `Application` class's abstract `start` method?

12.13 The program in Code Listing 12-1 calls a `Stage` class's `setTitle` method. What does this method do?

12.14 The program in Code Listing 12-1 calls a `Stage` class's `show` method. What does this method do?

## 12.3 Creating Scenes

**CONCEPT: A scene is a collection of controls and other objects in a JavaFX GUI.**

VideoNote
Creating Scenes

Recall that we use a theater metaphor when describing a JavaFX GUI. In JavaFX terminology, a window is called a stage, and the collection of GUI objects that is displayed in that window is called a scene. The program you saw in Code Listing 12-1 created a stage, but it did not create a scene. As a result, the program displays an empty window. Now, it's time to learn how to create a scene that contains JavaFX controls.

In general, the steps that you follow when creating a scene are:

1. Create the controls that will be in the scene.
2. Create a layout container of some type, and add the controls to the layout container.
3. Create a `Scene` object, and add the layout container to the `Scene` object.
4. Add the `Scene` object to the stage.

Let's take a closer look at each step.

## Creating Controls

The JavaFX library provides classes for all of the objects you can add to a GUI. The classes are in various subpackages of the `javafx` package, so you must write an `import` statement for the class you want to work with. For example, if you want to create a `Label` control, you will import the `Label` class, which is in the `javafx.scene.control` package.

Once you have written the necessary `import` statement, you will create an instance of the class. This is often done in the `start` method. For example, the following statement creates a `Label` control that displays the text *Hello World*:

```
Label messageLabel = new Label("Hello World");
```

Notice we pass the string we want displayed in the `Label` as an argument to the `Label` class's constructor. The `Label` class has multiple constructors, but this is the one you will use most often.

Regardless of the type of control you are creating, the process is typically the same: You import the class for the control, then in the `start` method you instantiate the class, calling the desired constructor.

## Creating Layout Containers

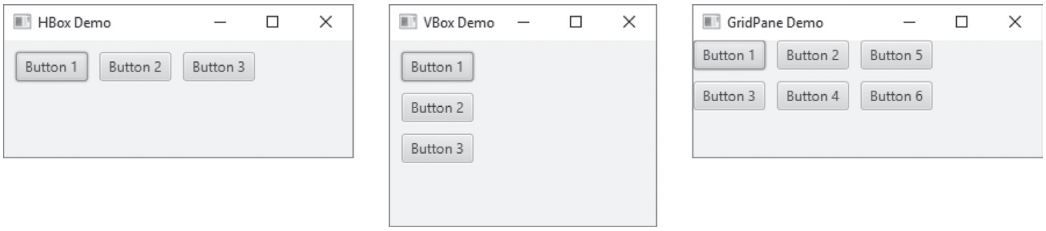
*Layout containers* are an important part of a JavaFX GUI. You use layout containers to arrange the positions of controls on the screen. JavaFX provides several different layout containers for you to choose from, and later in this chapter, we will look at many of them. For now, we will use only the three simple containers listed in Table 12-1.

**Table 12-1** Three of the most simple layout containers

| Layout Container | Description |
|---|---|
| `HBox` | Arranges controls in a single horizontal row. |
| `VBox` | Arranges controls in a single vertical row. |
| `GridPane` | Arranges controls in a grid with rows and columns. |

Figure 12-8 shows an example of how the three layout containers listed in Table 12-1 affect the arrangement of controls. The leftmost screen shows an application with three `Button` controls in an `HBox`. The middle screen shows an application with three `Button` controls in a `VBox`. The rightmost screen shows an application with six `Button`s in a `GridPane`.

**Figure 12-8** Examples of layout containers (Oracle Corporate Counsel)



You create layout containers in the same manner that you create other GUI objects. First, you import the container's class from the necessary package. Then, you create an instance of the container's class, calling the desired constructor.

For example, if you want to create an `HBox`, you import the `HBox` class from the `javafx.scene.layout` package. Then, you create an instance of the `HBox` class. The following code snippet shows an example:

```
// Create a Label control.
Label messageLabel = new Label("Hello World");

// Create an HBox container and add the Label.
HBox hbox = new HBox(messageLabel);
```

First, this code creates a `Label` control. Then, it creates an `HBox` layout container, and adds the `Label` control to the `HBox`.

You can add multiple controls to an `HBox` by passing multiple arguments to the `HBox` constructor. For example, the following code snippet creates two `Label` controls, then adds them both to an `HBox`:

```
// Create two Label controls.
Label message1 = new Label("Hello");
Label message2 = new Label("World");

// Create an HBox container and add the Labels.
HBox hbox = new HBox(message1, message2);
```

This gives you a general idea of how to create a layout container, using `HBox` as an example. We will look at layout containers, including `HBox`, in greater detail as we progress through this chapter.

## Creating a Scene Object

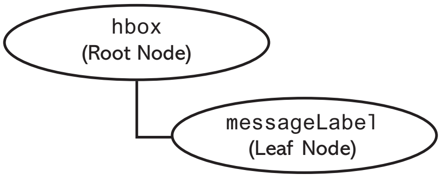
Once you have created the controls that you want in your scene, and added them to a layout container, the next step is to create an instance of the `Scene` class (which is in the `javafx.scene` package), and add the layout container to the `Scene` object.

The following code snippet shows an example. It creates a `Label` control, adds the `Label` to an `HBox` container, then adds the `HBox` container to a `Scene` object:

```
// Create a Label control.
Label messageLabel = new Label("Hello World");

// Create an HBox container and add the Label.
HBox hbox = new HBox(messageLabel);

// Create a Scene and add the HBox as the root node.
Scene scene = new Scene(hbox);
```

The argument we pass to the `Scene` constructor is the scene's *root node*. In this example, the `HBox` container will become the root node of the scene graph, as shown in Figure 12-9.

**Figure 12-9** Nodes in the scene graph



## Adding the `Scene` Object to the Stage

Once the `Scene` object has been created, the next step is to add the `Scene` object to the stage. Recall the `start` method receives a reference to a `Stage` object as an argument. To add your `Scene` object to the stage, call the `Stage` object's `setStage` method, passing a reference to your `Scene` object as an argument.

Let's assume `primaryStage` references the `Stage` object, and `scene` references your `Scene` object. Here is an example of how you would call the `setScene` method to add the scene to the stage:

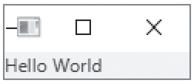
```
primaryStage.setScene(scene);
```

Now that we've covered the basics, we can look at a complete program that creates a simple scene. Look at Code Listing 12-2, a simple Hello World application. This application's scene contains an `HBox` with a `Label` displaying the text *Hello World*. When you compile and execute this program, you will see a window similar to the one shown in Figure 12-10.

**Code Listing 12-2** `(HelloWorld.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;

/**
   A JavaFX Hello World application
*/

public class HelloWorld extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
21       {
22          // Create a Label control.
23          Label messageLabel = new Label("Hello World");
24
25          // Put the Label in an HBox.
26          HBox hbox = new HBox(messageLabel);
27
28          // Create a Scene with the HBox as its root node.
29          Scene scene = new Scene(hbox);
30
31          // Add the Scene to the Stage.
32          primaryStage.setScene(scene);
33
34          // Set the stage title.
35          primaryStage.setTitle("My First Scene");
36
37          // Show the window.
38          primaryStage.show();
39       }
40    }
```

**Figure 12-10** Window displayed by the HelloWorld application (Oracle Corporate Counsel)



Hello World

Let's take a closer look at the program in Code Listing 12-2:

- Lines 1 through 5 import the following JavaFX classes: `Application`, `Stage`, `Scene`, `HBox`, and `Label`.
- The declaration for the `HelloWorld` class begins in line 11. Notice it extends the `Application` class.
- The `main` method appears in lines 13 through 17. The `main` method does only one thing: it calls the `launch` method in line 16. The `launch` method, which is inherited from the `Application` class, does a number of setup operations, including the following:
  - It creates a `Stage` object that will be the application's window.
  - It calls the `start` method, passing a reference to the `Stage` object as an argument.
- The `start` method appears in lines 20 through 39. This is the entry point for the application. The `start` method is an abstract method in the `Application` class, and we must override it. Notice the method has a parameter named `primaryStage`. The `primaryStage` variable will reference the `Stage` object created by the `launch` method.
- Line 23 creates a `Label` control named `messageLabel` that displays the string `"Hello World"`.
- Line 26 creates an `HBox` container named `hbox`, and adds the `messageLabel` control to it.
- Line 29 creates a `Scene` object named `scene`, and adds `hbox` as the scene's root node.
- Line 32 calls the `primaryStage` object's `setScene` method to set the scene to the stage.

- Line 35 calls the `primaryStage` object's `setTitle` method. This sets the text that will be displayed in the window's title bar.
- Line 38 calls the `primaryStage` object's `show` method. This displays the application's window.

## Setting the Size of the Scene

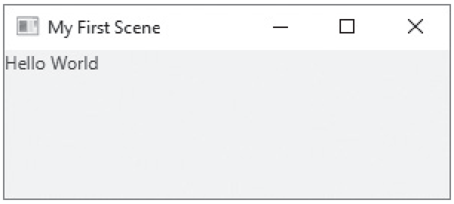
Notice in Figure 12-10 that the window is very small. In fact, it is just large enough to display the `Label` control and the standard Windows buttons in the title bar. By default, when you create a `Scene` object, the scene will be just large enough to display the contents of the scene. This is known as the scene's *preferred size*.

If you wish, you can specify the size of the scene by passing width and height arguments to the `Scene` class constructor. For example, suppose we modify line 29 of Code Listing 12-2 as follows:

```
Scene scene = new Scene(hbox, 300, 100);
```

The first argument, `hbox`, is the root node. The second argument, 300, is the scene's width, in pixels. The third argument, 100, is the scene's height, in pixels. As a result of adding these width and height arguments to the constructor call, the program will display a window similar to Figure 12-11.

**Figure 12-11** Window displayed by the modified HelloWorld application (Oracle Corporate Counsel)



Hello World

## Aligning Controls in an `HBox` Layout Container

Notice in Figure 12-11 that the `Label` control is displayed in the top-left corner of the scene. That is the default alignment of an `HBox` layout container. You can change the alignment of an `HBox` by calling its `setAlignment` method. For example, assume `hbox` references an `HBox` object. The following statement causes all controls in the `hbox` container to be centered within the container:

```
hbox.setAlignment(Pos.CENTER);
```

The argument you pass to the `setAlignment` method is one of the `enum` constants listed in Table 12-2. The `Pos` type must be imported from the `javafx.geometry` package.

**Table 12-2** enum constants of the Pos type

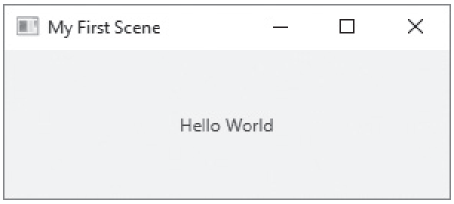
| | | |
|---|---|---|
| `Pos.TOP_LEFT` | `Pos.TOP_CENTER` | `Pos.TOP_RIGHT` |
| `Pos.CENTER_LEFT` | `Pos.CENTER` | `Pos.CENTER_RIGHT` |
| `Pos.BOTTOM_LEFT` | `Pos.BOTTOM_CENTER` | `Pos.BOTTOM_RIGHT` |

For example, Code Listing 12-3 shows a modified version of the `HelloWorld` program that you previously saw in Code Listing 12-2. In this version, the `messageLabel` control is centered in an `HBox` that is 300 pixels wide and 100 pixels high. When you compile and execute this program, you will see a window similar to the one shown in Figure 12-12.

**Code Listing 12-3** **(`HelloWorld2.java`)**

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;
import javafx.geometry.Pos;

/**
   A JavaFX Hello World application
*/

public class HelloWorld2 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label control.
      Label messageLabel = new Label("Hello World");

      // Put the Label in an HBox.
      HBox hbox = new HBox(messageLabel);

      // Create a Scene with the HBox as its root node.
      Scene scene = new Scene(hbox, 300, 100);

      // Set the HBox's alignment to center.
      hbox.setAlignment(Pos.CENTER);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Set the stage title.
      primaryStage.setTitle("My First Scene");
40
41         // Show the window.
42         primaryStage.show();
43      }
44   }
```

**Figure 12-12** Window displayed by the `HelloWorld2` application (Oracle Corporate Counsel)



Notice in line 6, we imported the `Pos` enumeration from the `javafx.geometry` package. In line 30, we create a `Scene` object that is 300 pixels wide by 100 pixels high, with the `hbox` object as the root node. Then, in line 33, we call the `hbox` object's `setAlignment` method, passing `Pos.CENTER` as an argument.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.15 What is the general difference between an `HBox` layout container, and a `VBox` layout container?

12.16 If you want to arrange controls in a grid, with rows and columns, what layout container would you use?

12.17 The `Scene` class is in what package?

12.18 How do you change the alignment of an `HBox` container?

12.19 What package is the `Pos` enumeration in?

## 12.4 Displaying Images

**CONCEPT: You use both the `Image` and `ImageView` classes to display an image.**

VideoNote
Displaying Images

Displaying an image in a JavaFX application is a two-step process: First, you load the image into memory, then second, you display the image. This requires two classes from the JavaFX library: `Image` and `ImageView`. Both of these classes are in the `javafx.scene.image` package.

The `Image` class can load an image from the computer's local file system, or from an Internet location. The class supports the BMP, JPEG, GIF, and PNG file types. To load an image of any of these types, create an instance of the `Image` class, passing the constructor a string argument that specifies the file's name and location. Here is an example:

```
Image image = new Image("file:HotAirBalloon.jpg");
```

In this example, the string that we are passing to the constructor specifies a file named HotAirBalloon.jpg. The part of the string that reads `file:` is a protocol identifier indicating that the file is on the local computer. Because no path was given, it is assumed the file is in the same directory or folder as the program's executable .class file. If you want to load an image from a different location, you can specify a path. Here is an example:

```
Image image = new Image("file:C:\\Images\\HotAirBalloon.jpg");
```

Next, you create an instance of the `ImageView` class, passing a reference to the `Image` object as an argument to the constructor. Here is an example:

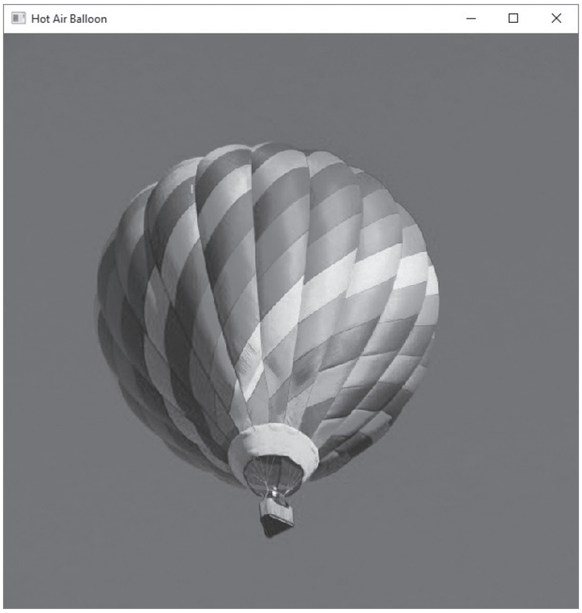
```
ImageView imageView = new ImageView(image);
```

Once you've created an `ImageView` object, you can add it to a layout container. The program in Code Listing 12-4 shows an example. Figure 12-13 shows the program's output.

**Code Listing 12-4** (`ImageDemo.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

/**
  An Image Demo
*/

public class ImageDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create an Image object.
      Image image = new Image("file:HotAirBalloon.jpg");

      // Create an ImageView object.
      ImageView imageView = new ImageView(image);

      // Put the ImageView in an HBox.
      HBox hbox = new HBox(imageView);

      // Create a Scene with the HBox as its root node.
33          Scene scene = new Scene(hbox);
34
35          // Add the Scene to the Stage.
36          primaryStage.setScene(scene);
37
38          // Set the stage title.
39          primaryStage.setTitle("Hot Air Balloon");
40
41          // Show the window.
42          primaryStage.show();
43       }
44    }
```

**Figure 12-13** The ImageDemo application

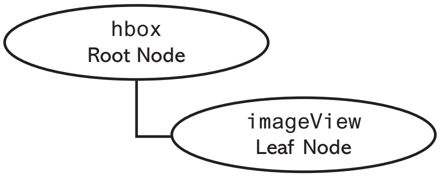


Let's take a closer look at the code:

- Lines 5 and 6 import the `Image` class and the `ImageView` class from the `javafx.scene.image` package.
- Line 24 creates an `Image` object that loads the file HotAirBalloon.jpg from the local file system.
- Line 27 creates an `ImageView` object, using the `Image` object created in line 24.
- Line 30 creates an `HBox`, adding the `ImageView` object to it.
- Line 33 creates a `Scene` object with the `HBox` as the root node.
- Line 36 sets the scene to the stage.
- Line 39 sets the text to be displayed in the window's title bar.
- Line 42 displays the application's window.

Figure 12-14 depicts a scene graph for Code Listing 12-4.

**Figure 12-14** Scene graph for the ImageDemo program in Code Listing 12-4



> **NOTE:** Notice in Figure 12-14 that the `ImageView` object is a node in the scene graph, but the `Image` object is not. The `Image` class loads an image into memory, but it does not display the image on the screen. Displaying an image is the job of the `ImageView` class. Therefore, the `ImageView` object is a node in the scene graph, but the `Image` object is not.

## Loading Images from an Internet Location

The string that you pass as an argument to the `Image` class constructor specifies a protocol used to load the image. The example you saw in Code Listing 12-4 uses the `file:` protocol to load the image from the local file system. If you want to load an image from an Internet location, you can use the `http:` protocol, as shown in the following example:

```
Image image = new Image("http://www.gaddisbooks.com/images/HotAirBalloon.jpg");
```

## Setting the Size of an Image

By default, the `ImageView` class displays an image at its full size. You can resize the image with the `ImageView` class's `setFitWidth` and `setFitHeight` methods. Let's assume `myImageView` references an `ImageView` object. The following code shows an example:

```
myImageView.setFitWidth(100);
myImageView.setFitHeight(100);
```

After this code executes, the image size of the `ImageView` object will be set to 100 pixels wide by 100 pixels high.

> **NOTE:** By default, if you set the scene's size to a width and height that is different from the image's width and height, the image will *not* be scaled to fit the size of the scene. If the image is smaller than the scene, the image will occupy only part of the window. If the image is larger than the program's window, only part of the image will be displayed.

## Preserving the Image's Aspect Ratio

Usually when you resize an image, you want to preserve the image's aspect ratio. The *aspect ratio* is the ratio of the image's width to the image's height. If you change an image's aspect ratio, the image will appear stretched (either horizontally or vertically).

The `ImageView` class has a method named `setPreserveRatio` that you can call to make sure an image's aspect ratio is preserved. Let's assume that `myImageView` references an `ImageView` object. The following code shows an example:

```
myImageView.setPreserveRatio(true);
```

The `setPreserveRatio` method accepts a Boolean argument. If you pass `true` as an argument, the image's aspect ratio is preserved as follows:

- If you set both the image's width and height, the width and/or the height may be scaled to get the closest possible fit while also preserving the aspect ratio.
- If you set only the image's width, the height will be scaled to preserve the aspect ratio.
- If you set only the image's height, the width will be scaled to preserve the aspect ratio.

If you pass `false` as an argument to the `setPreserveRatio` method, it is the same as not calling the method at all. As a result, any changes you make to the image's width and/or height will not preserve the image's aspect ratio. This will potentially cause the image to appear stretched in one direction.

### Changing an ImageView's Image

You can change the image that is displayed by an `ImageView` object by calling the object's `setImage` method. The `setImage` method takes, as an argument, a reference to an `Image` object you want to display. For example, let's assume `myImageView` references an `ImageView` object, and `myImage` references an `Image` object. The following code shows an example of calling the `setImage` method:

```
myImageView.setImage(myImage);
```

After this statement executes, the `myImageView` object will display the image that was loaded by the `myImage` object.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.20 What package contains the `Image` and `ImageView` classes?

12.21 If you want to display an image in a layout container, do you add an `Image` object, or an `ImageView` object to the layout container?

12.22 How do you set the width and height of an `ImageView`?

12.23 If you want to make sure that an image does not appear stretched after it is resized, what method do you call?

12.24 If you want to change the image that an `ImageView` object is displaying, what method do you call?

## 12.5 More about the HBox, VBox, and GridPane Layout Containers

**CONCEPT: The layout containers provide methods for fine-tuning the way controls are arranged.**

We have briefly introduced the `HBox`, `VBox`, and `GridPane` layout containers. In this section, we will look more closely at each of these layout containers, and discuss ways to fine-tune the ways in which they arrange controls.

## The HBox Layout Container

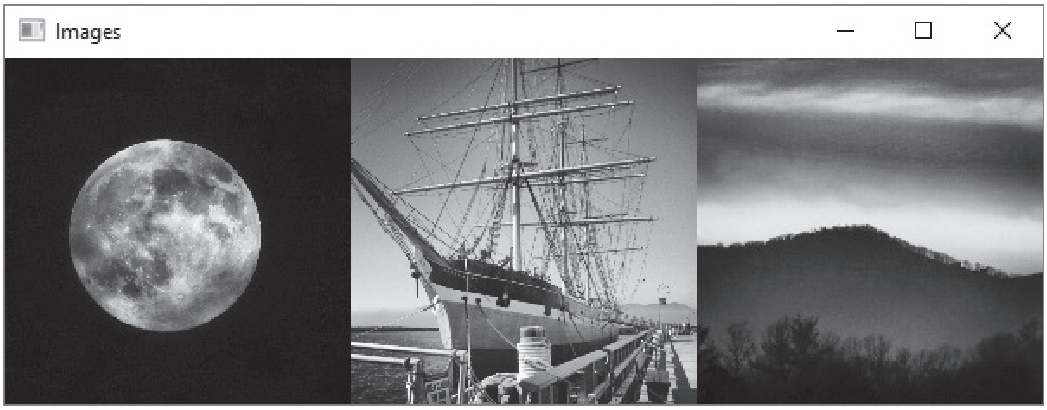
VideoNote The HBox Layout Container

You have already seen programs that use the HBox layout container, which arranges controls in a single horizontal row. You can add one or more controls to an HBox by passing references to those controls as arguments to the HBox constructor. For example, the program in Code Listing 12-5 displays three images in an HBox. The program's output is shown in Figure 12-15.

**Code Listing 12-5** (HBoxImages.java)

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

/**
  This program displays images in an Hbox.
*/

public class HBoxImages extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create the Image objects.
      Image moonImage = new Image("file:Moon.jpg");
      Image shipImage = new Image("file:Ship.jpg");
      Image sunsetImage = new Image("file:Sunset.jpg");

      // Create the ImageView objects.
      ImageView moonIView = new ImageView(moonImage);
      ImageView shipIView = new ImageView(shipImage);
      ImageView sunsetIView = new ImageView(sunsetImage);

      // Resize the moon image, preserving its aspect ratio.
      moonIView.setFitWidth(200);
      moonIView.setPreserveRatio(true);

      // Resize the ship image, preserving its aspect ratio.
      shipIView.setFitWidth(200);
      shipIView.setPreserveRatio(true);
40
41           // Resize the sunset image, preserving its aspect ratio.
42           sunsetIView.setFitWidth(200);
43           sunsetIView.setPreserveRatio(true);
44
45           // Put the ImageViews in an HBox.
46           HBox hbox = new HBox(moonIView, shipIView, sunsetIView);
47
48           // Create a Scene with the HBox as its root node.
49           Scene scene = new Scene(hbox);
50
51           // Add the Scene to the Stage.
52           primaryStage.setScene(scene);
53
54           // Set the stage title.
55           primaryStage.setTitle("Images");
56
57           // Show the window.
58           primaryStage.show();
59       }
60   }
```

**Figure 12-15** The HBoxImages application



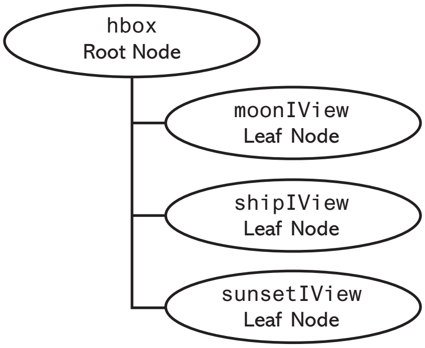
Let's take a closer look at the code:

- Lines 24 through 26 create three `Image` objects, and load the image files Moon.jpg, Ship.jpg, and Sunset.jpg.
- Lines 29 through 31 create three `ImageView` objects for the image files that were loaded. The names of the `ImageFile` objects are `moonIView`, `shipIView`, and `sunsetIView`.

- Line 34 sets the width of the `moonIView` object to 200 pixels. Line 35 causes the aspect ratio to be preserved. As a result, the image's height will be automatically scaled so the image does not appear stretched.
- Line 38 sets the width of the `shipIView` object to 200 pixels. Line 39 causes the aspect ratio to be preserved. As a result, the image's height will be automatically scaled so the image does not appear stretched.
- Line 42 sets the width of the `sunsetIView` object to 200 pixels. Line 43 causes the aspect ratio to be preserved. As a result, the image's height will be automatically scaled so the image does not appear stretched.
- Line 46 creates an `HBox` container, and adds the `moonIView`, `shipIView`, and `sunsetIView` objects to it.
- Line 49 creates a `Scene` object with the `HBox` as the root node.
- Line 52 sets the scene to the stage.
- Line 55 sets the text that will be displayed in the window's title bar.
- Line 58 displays the application's window.

Figure 12-16 depicts a scene graph for Code Listing 12-5.

**Figure 12-16** Scene graph for the HBoxImages program in Code Listing 12-5



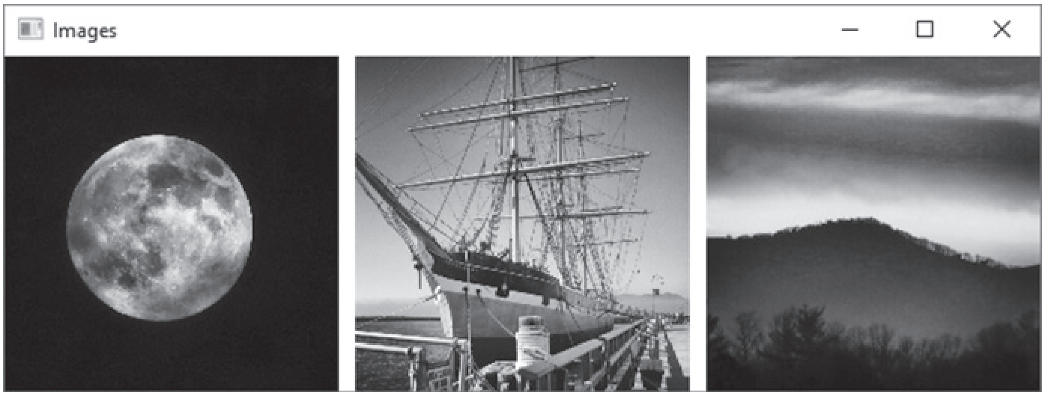
### Spacing

Notice in Figure 12-15, the images are displayed with no space between them. If we want some space to appear between the controls in an `HBox`, we can optionally pass a spacing value as the first argument to the `HBox` constructor. Here is an example of how we might have written the statement in line 46 of Code Listing 12-5:

```
HBox hbox = new HBox(10, moonIView, shipIView, sunsetIView);
```

If the first argument passed to the `HBox` constructor is a number, that value is used as the number of pixels to appear between the controls horizontally in the container. In this example, the images will be displayed with 10 pixels of space between them. This will cause the program's window to appear as shown in Figure 12-17.

**Figure 12-17** Images in an HBox with 10 pixels spacing



### Padding

*Padding* is space that appears around the inside edge of a container. If you want to add padding to an `HBox`, you call the `HBox` object's `setPadding` method. The `setPadding` method takes an `Insets` object as its argument. When constructing the `Insets` object, pass a number of pixels as an argument to the constructor. For example, the following statement adds 10 pixels of padding to an `HBox` container named `hbox`:

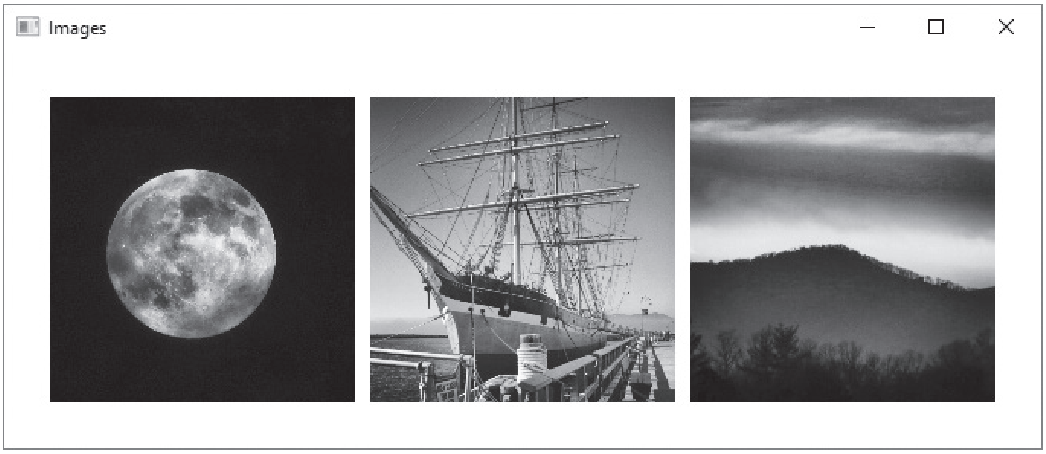
```
hbox.setPadding(new Insets(10));
```

The `Insets` class is in the `javafx.geometry` package, so be sure to include the necessary `import` statement in your program. Code Listing 12-6 shows an example. The program's output is shown in Figure 12-18.

**Code Listing 12-6** (`HBoxImagesWithPadding.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.geometry.Insets;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

/**
  This program demonstrates padding and spacing.
*/

public class HBoxImagesWithPadding extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create the Image components.
      Image moonImage = new Image("file:Moon.jpg");
      Image shipImage = new Image("file:Ship.jpg");
      Image sunsetImage = new Image("file:Sunset.jpg");

      // Create the ImageView components.
      ImageView moonIView = new ImageView(moonImage);
      ImageView shipIView = new ImageView(shipImage);
      ImageView sunsetIView = new ImageView(sunsetImage);

      // Resize the moon image, preserving its aspect ratio.
      moonIView.setFitWidth(200);
      moonIView.setPreserveRatio(true);

      // Resize the ship image, preserving its aspect ratio.
      shipIView.setFitWidth(200);
      shipIView.setPreserveRatio(true);

      // Resize the sunset image, preserving its aspect ratio.
      sunsetIView.setFitWidth(200);
      sunsetIView.setPreserveRatio(true);

      // Put the ImageViews in an HBox with 10 pixels spacing.
      HBox hbox = new HBox(10, moonIView, shipIView, sunsetIView);

      // Put 30 pixels of padding around the HBox.
      hbox.setPadding(new Insets(30));

      // Create a Scene with the HBox as its root node.
      Scene scene = new Scene(hbox);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Set the stage title.
      primaryStage.setTitle("Images");

      // Show the window.
      primaryStage.show();
   }
}
```

**Figure 12-18** The HBoxImagesWithPadding application



This program is a modification of the program you saw in Code Listing 12-5. Here are the differences:

- Line 5 imports the `Insets` class from the `javafx.geometry` class.
- Line 47 creates an `HBox` container with 10 pixels of horizontal spacing between the images.
- Line 50 sets the `HBox` container's padding to 30 pixels.

## The `VBox` Layout Container

**VideoNote** The VBox Layout Container

The `VBox` layout container arranges controls in a single vertical row. If you want to create a `VBox`, you import the `VBox` class from the `javafx.scene.layout` package. Then you create an instance of the `VBox` class. The following code snippet shows an example:

```
// Create a Label control.
Label messageLabel = new Label("Hello World");

// Create a VBox container and add the Label.
VBox vbox = new VBox(messageLabel);
```

First, this code creates a `Label` control. Then, it creates a `VBox` layout container, and adds the `Label` control to the `VBox`.

You can add multiple controls to a `VBox` by passing multiple arguments to the `VBox` constructor. For example, the following code snippet creates two `Label` controls, then adds them both to a `VBox`:

```
// Create two Label controls.
Label message1 = new Label("Hello");
Label message2 = new Label("World");

// Create a VBox container and add the Labels.
VBox vbox = new VBox(message1, message2);
```

By default, items appear in a `VBox` with no space between them. If you want some space to appear between the objects in a `VBox`, you can optionally pass a spacing value as the first argument to the `VBox` constructor. Here is an example:

```
VBox vbox = new VBox (10, message1, message2);
```

If the first argument passed to the `VBox` constructor is a number, that value is used as the number of pixels to appear horizontally between the controls in the container. In this example, the `Labels` will be displayed with 10 pixels of space between them.

Recall that padding is space that appears around the inside edge of a container. By default, there is no padding in a `VBox`. If you want to add padding to a `VBox`, you call the `VBox` object's `setPadding` method. The `setPadding` method takes an `Insets` object as its argument. When constructing the `Insets` object, pass a number of pixels as an argument to the constructor. For example, the following statement adds 10 pixels of padding to a `VBox` container named `vbox`:

```
vbox.setPadding(new Insets(10));
```

Recall that the `Insets` class is in the `javafx.geometry` package, so be sure to include the necessary `import` statement in your program.

The program in Code Listing 12-7 shows an example of how to use the `VBox` layout container. It is very similar to Code Listing 12-6, except that it displays the three images in a `VBox`. The program's output is shown in Figure 12-19. Figure 12-20 depicts the application's scene graph.

**Code Listing 12-7** (`VBoxImagesWithPadding.java`)

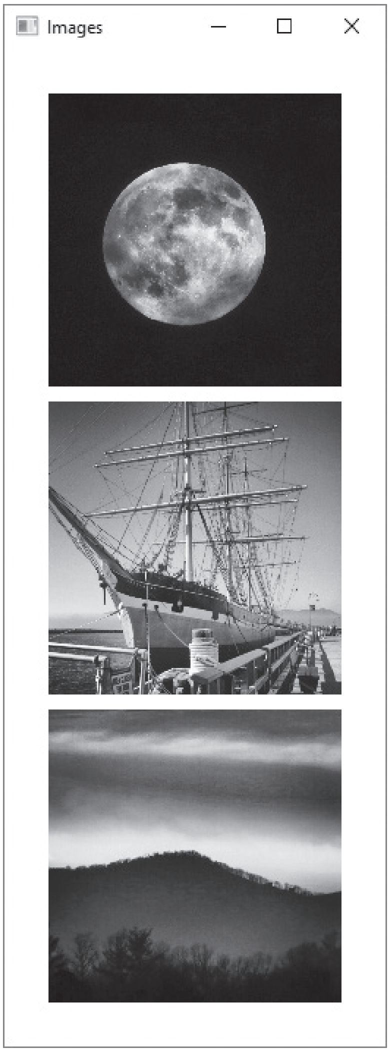
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.geometry.Insets;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

/**
  This program demonstrates the VBox layout container.
*/

public class VBoxImagesWithPadding extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create the Image components.
      Image moonImage = new Image("file:Moon.jpg");
      Image shipImage = new Image("file:Ship.jpg");
      Image sunsetImage = new Image("file:Sunset.jpg");

      // Create the ImageView components.
      ImageView moonIView = new ImageView(moonImage);
      ImageView shipIView = new ImageView(shipImage);
      ImageView sunsetIView = new ImageView(sunsetImage);

      // Resize the moon image, preserving its aspect ratio.
      moonIView.setFitWidth(200);
      moonIView.setPreserveRatio(true);

      // Resize the ship image, preserving its aspect ratio.
      shipIView.setFitWidth(200);
      shipIView.setPreserveRatio(true);

      // Resize the sunset image, preserving its aspect ratio.
      sunsetIView.setFitWidth(200);
      sunsetIView.setPreserveRatio(true);

      // Put the ImageViews in a VBox with 10 pixels spacing.
      VBox vbox = new VBox(10, moonIView, shipIView, sunsetIView);

      // Put 30 pixels of padding around the VBox.
      vbox.setPadding(new Insets(30));

      // Create a Scene with the VBox as its root node.
      Scene scene = new Scene(vbox);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Set the stage title.
      primaryStage.setTitle("Images");

      // Show the window.
      primaryStage.show();
   }
}
```

## The `GridPane` Layout Container

VideoNote
The `GridPane` Layout Container

The `GridPane` layout container arranges its contents in a grid with columns and rows. As a result, a `GridPane` is divided into cells, much like a spreadsheet. As shown in Figure 12-21, the columns and rows are identified by indexes, beginning at 0. You use these column and row indexes to refer to specific cells within the `GridPane`.

**Figure 12-19** The VBoxImagesWithPadding application



If you want to create a `GridPane`, you import the `GridPane` class from the `javafx.scene.layout` package. Then, you create an instance of the `GridPane` class using its no-arg constructor, as shown here:

```
GridPane gridpane = new GridPane();
```

Once you have created a `GridPane` object, you can call the object's `add` method to add controls at specific positions in the grid. Here is the general format of the `add` method:

```
gridPaneObject.add(control, column, row);
```

In the general format, ***gridPaneObject*** is a reference to a `GridPane`, ***control*** is the control you are adding to the `GridPane`, ***column*** is the column index, and ***row*** is the row index.

The program in Code Listing 12-8 demonstrates how to use a `GridPane`. The program's output is shown in Figure 12-22.

**Figure 12-20** Scene graph for the VBoxImagesWithPadding program in Code Listing 12-7

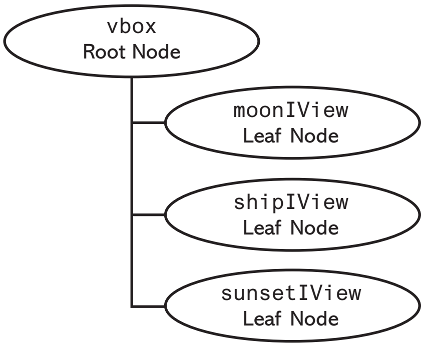
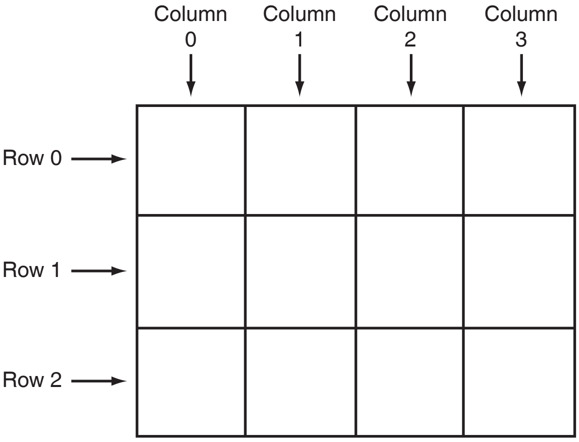

vbox
Root Node
moonIView
Leaf Node
shipIView
Leaf Node
sunsetIView
Leaf Node

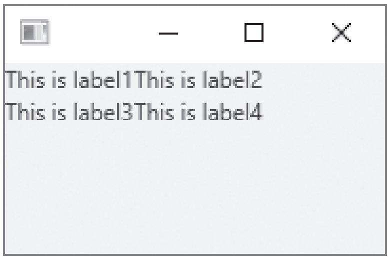
**Figure 12-21** Column and row indexes


Column 0
Column 1
Column 2
Column 3
Row 0
Row 1
Row 2

**Code Listing 12-8** (`GridPaneDemo.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.GridPane;
import javafx.geometry.Insets;
import javafx.scene.control.Label;

/**
   GridPane Demo
*/

public class GridPaneDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create some Label controls.
      Label label1 = new Label("This is label1");
      Label label2 = new Label("This is label2");
      Label label3 = new Label("This is label3");
      Label label4 = new Label("This is label4");

      // Create a GridPane.
      GridPane gridpane = new GridPane();

      // Add the Labels to the GridPane.
      gridpane.add(label1, 0, 0); // Column 0, Row 0
      gridpane.add(label2, 1, 0); // Column 1, Row 0
      gridpane.add(label3, 0, 1); // Column 0, Row 1
      gridpane.add(label4, 1, 1); // Column 1, Row 1

      // Create a Scene with the GridPane as its root node.
      // The Scene is 200 pixels wide by 100 pixels high.
      Scene scene = new Scene(gridpane, 200, 100);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Show the window.
      primaryStage.show();
   }
}
```

**Figure 12-22** The GridPaneDemo application (Oracle Corporate Counsel)

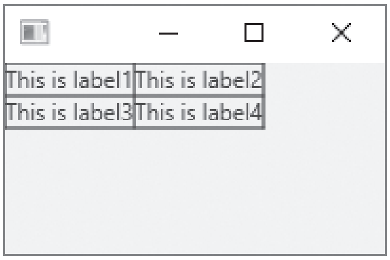


```
This is label1This is label2
This is label3This is label4
```

The `GridPane` class has a method named `setGridLinesVisible` that is helpful for debugging purposes. When you call the method, passing true as its argument, the `GridPane` is displayed with grid lines. For example, if we added the following statement to Code Listing 12-8, the program's output would appear as shown in Figure 12-23:

```
gridpane.setGridLinesVisible(true);
```

**Figure 12-23** A GridPane with visible grid lines (Oracle Corporate Counsel)



| | |
|---|---|
| This is label1 | This is label2 |
| This is label3 | This is label4 |

By default, there is no space between the columns and rows in a `GridPane`. If you want space to appear between the columns in a `GridPane`, you can call the `GridPane` class's `setHgap` method. The argument you pass to the `setHgap` method is the number of pixels of space you want between the columns. Let's assume `gridpane` references a `GridPane` object. The following statement shows an example of calling the `setHgap` method:

```
gridpane.setHgap(10);
```

This statement will cause 10 pixels of space to appear between the columns of the `GridPane`. If you want vertical space to appear between the rows in a `GridPane`, you can call the `GridPane` class's `setVgap` method. Once again, assume `gridpane` references a `GridPane` object. The following statement calls the `setVgap` method to display 10 pixels of space between the rows of the `GridPane` object:

```
gridpane.setVgap(10);
```

By default, there is no padding around the inside edge of a `GridPane`. However, you can add padding by calling the `GridPane` object's `setPadding` method. Recall that the `setPadding` method takes an `Insets` object as its argument. When constructing the `Insets` object, pass a number of pixels as an argument to the constructor. For example, the following statement adds 10 pixels of padding to a `GridPane` container named `gridpane`:

```
gridpane.setPadding(new Insets(10));
```

Recall that the `Insets` class is in the `javafx.geometry` package, so be sure to include the necessary `import` statement in your program.

Code Listing 12-9 shows a program that displays images in a `GridPane` with 4 columns and 2 rows. The `GridPane` has 10 pixels of space between the cells, and 30 pixels of padding. The program's output is shown in Figure 12-24. Figure 12-25 depicts the application's scene graph.

**Code Listing 12-9** (`GridPaneImages.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.GridPane;
import javafx.geometry.Insets;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

/**
  This program demonstrates the GridPane layout container.
*/

public class GridPaneImages extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create the Image components.
      Image moonImage = new Image("file:Moon.jpg");
      Image shipImage = new Image("file:Ship.jpg");
      Image sunsetImage = new Image("file:Sunset.jpg");
      Image flowerImage = new Image("file:Flower.jpg");

      // Create the ImageView components.
      ImageView moonIView = new ImageView(moonImage);
      ImageView shipIView = new ImageView(shipImage);
      ImageView sunsetIView = new ImageView(sunsetImage);
      ImageView flowerIView = new ImageView(flowerImage);

      // Resize the moon image, preserving its aspect ratio.
      moonIView.setFitWidth(200);
      moonIView.setPreserveRatio(true);

      // Resize the ship image, preserving its aspect ratio.
      shipIView.setFitWidth(200);
      shipIView.setPreserveRatio(true);

      // Resize the sunset image, preserving its aspect ratio.
      sunsetIView.setFitWidth(200);
      sunsetIView.setPreserveRatio(true);

      // Resize the flower image, preserving its aspect ratio.
      flowerIView.setFitWidth(200);
      flowerIView.setPreserveRatio(true);

      // Create a GridPane.
      GridPane gridpane = new GridPane();
```

```

        // Add the ImageViews to the GridPane.
        gridpane.add(moonIView, 0, 0);    // Col 0, Row 0
        gridpane.add(shipIView, 1, 0);    // Col 1, Row 0
        gridpane.add(sunsetIView, 0, 1);  // Col 0, Row 1
        gridpane.add(flowerIView, 1, 1);  // Col 1, Row 1

        // Set the gap sizes.
        gridpane.setVgap(10);
        gridpane.setHgap(10);

        // Set the GridPane's padding.
        gridpane.setPadding(new Insets(30));

        // Create a Scene with the GridPane as its root node.
        Scene scene = new Scene(gridpane);

        // Add the Scene to the Stage.
        primaryStage.setScene(scene);

        // Set the stage title.
        primaryStage.setTitle("Images");

        // Show the window.
        primaryStage.show();
    }
}
```

**Figure 12-24** The GridPaneImages application

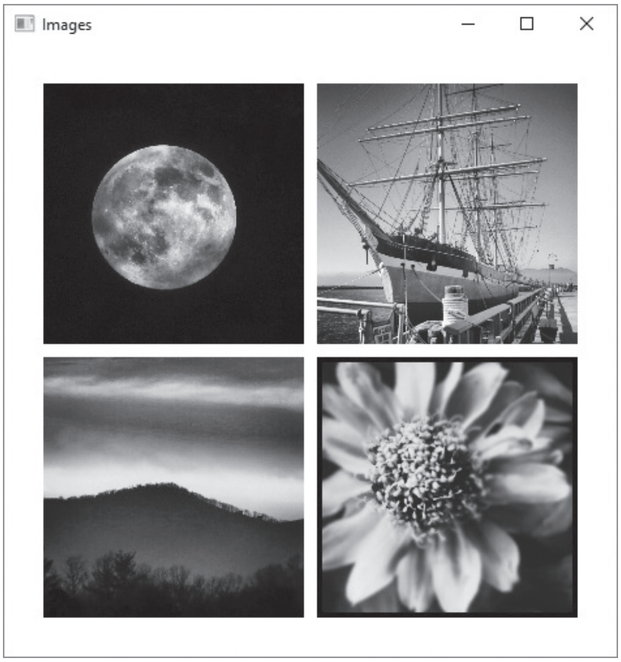
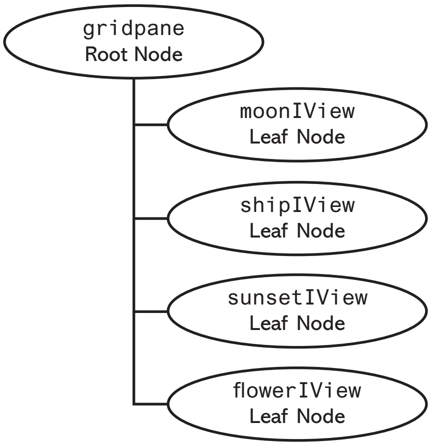


**Figure 12-25** Scene graph for the GridPaneImages program in Code Listing 12-9


gridpane
Root Node
moonIView
Leaf Node
shipIView
Leaf Node
sunsetIView
Leaf Node
flowerIView
Leaf Node

## Using Multiple Layout Containers in the Same Screen

Quite often, to get the particular screen layout that you desire, you will have to use multiple layout containers. You can even nest a layout container inside of another layout container. For example, Figure 12-26 shows a window that has a `GridPane` container with a `VBox` container nested inside of it. Here are some details about the containers in the figure (as illustrated in Figure 12-27):

- The `GridPane` has two columns and one row.
- The `Label` control is in the `GridPane`'s left column, and the `VBox` is in the `GridPane`'s right column.
- The `ImageView` objects are in the `GridPane`.

**Figure 12-26** A window with nested layout containers (Oracle Corporate Counsel)

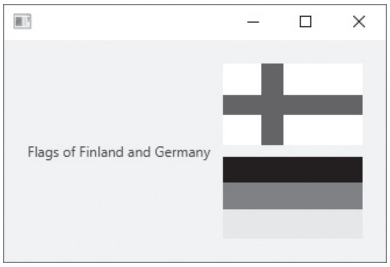
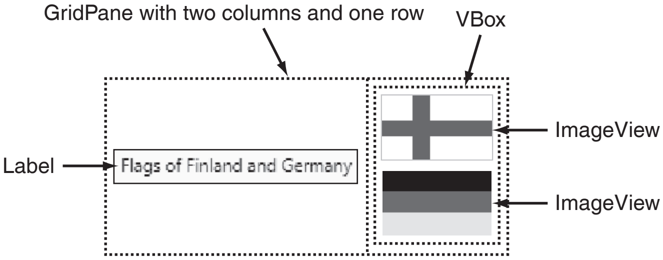

Flags of Finland and Germany

**Figure 12-27** The layout


GridPane with two columns and one row
VBox
ImageView
Label
Flags of Finland and Germany
ImageView

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.25 What is the difference between spacing and padding, in regard to layout containers?

12.26 What package is the `Insets` class in?

12.27 How do you set the horizontal and vertical spacing in a `GridPane` container?

12.28 Once you have created a `GridPane` container, how do you add a control to a specific column and row?

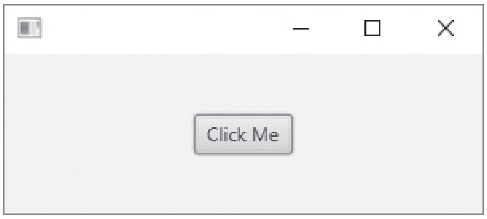
## 12.6 `Button` Controls and Events

**CONCEPT: A `Button` is a control the user can click to cause an action that you have programmed to take place.**

VideoNote
`Button` Controls and Events

A *`Button`* is a rectangular control that appears as a button with a caption written across its face. Typically, when the user clicks a `Button` control (either with a mouse, or by touching it on a touch screen), an action takes place. Figure 12-28 shows an example of a JavaFX window that contains a `Button` control.

**Figure 12-28** A window with a `Button` control (Oracle Corporate Counsel)



To create a `Button` control, you will use the `Button` class, which is in the `javafx.scene.control` package. Once you have written the necessary `import` statement, you will create an instance of the `Button` class. The following statement shows an example of creating a `Button` object that displays the text *Click Me*:

```
Button myButton = new Button("Click Me");
```

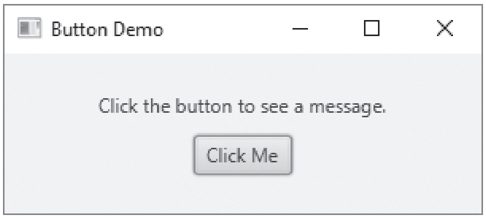
Notice that we pass the string we want displayed on the `Button` as an argument to the `Button` class's constructor. The `Button` class has multiple constructors, but this is the one you will use most often.

For example, look at Code Listing 12-10. When you compile and execute this program, you will see the window shown in Figure 12-29.

**Code Listing 12-10** (`ButtonDemo.java`)

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Pos;

/**
  A Button Demo
*/

public class ButtonDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label control.
      Label myLabel = new Label("Click the button to see a message.");

      // Create a Button control.
      Button myButton = new Button("Click Me");

      // Put the Label and Button in a VBox with 10 pixels of spacing.
      VBox vbox = new VBox(10, myLabel, myButton);

      // Create a Scene with the VBox as its root node.
      Scene scene = new Scene(vbox, 300, 100);

      // Set the scene's alignment to center.
      vbox.setAlignment(Pos.CENTER);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Set the stage title.
      primaryStage.setTitle("Button Demo");

45         // Show the window.
46         primaryStage.show();
47      }
48   }
```

**Figure 12-29** The ButtonDemo application (Oracle Corporate Counsel)



Click the button to see a message.

Here are some specific things to note about the `ButtonDemo` program:

- Line 6 imports the `Button` class from the `javafx.control` package.
- Line 28 creates an instance of the `Button` class. The resulting `Button` object is referenced by the `myButton` variable. The string `"Click Me"` is displayed on the `Button`.

If you compile and run this application, you will notice nothing happens when you click the `Button`. That requires some extra code. We will discuss that next, but for now, you can end the application by clicking the standard Close button in the window's title bar.

## Handling Events

An *event* is an action that takes place while a program is running. For example, each time the user clicks a `Button` control, an event takes place. When an event takes place, the control responsible for the event creates an *event object* that contains information about the event. The GUI control that created the event object is known as the *event source*.

Event objects are instances of the `Event` class (from the `javafx.event` package), or one of its subclasses. When a `Button` control is clicked, an event object that is an instance of the `ActionEvent` class is created. `ActionEvent` is a subclass of the `Event` class, and it is also in the `javafx.event` package.

But what happens to the event object once it is generated by an event source? It's possible the event source is connected to one or more event handlers. An *event handler* is an object that responds to events. If an event source is connected to an event handler, a specific method in the event handler is called and the event object is passed as an argument to the method. This process is sometimes referred to as *event firing*.

## Writing Event Handlers

When you are writing a GUI application, it is your responsibility to write the event handler classes that your application needs. When you write an event handler class, it must implement the `EventHandler` interface, which is in the `javafx.event` package. The `EventHandler` interface specifies a `void` method named `handle`. The `handle` method must have a parameter of the `Event` class, or one of its subclasses.

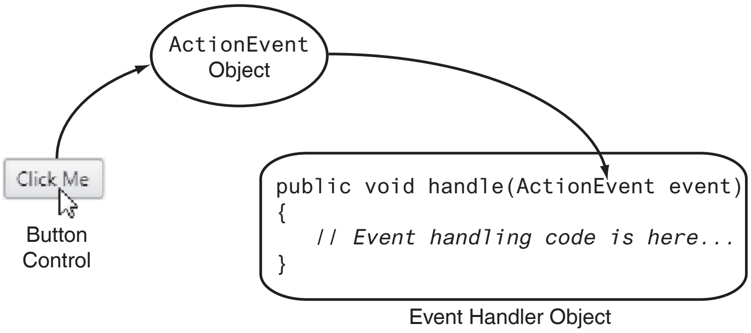
For example, suppose we want to write an event handler class that can respond to `Button` clicks. Recall that when a `Button` control is clicked, the event object that is generated is of the `ActionEvent` type. This is the general format of an event handler class that handles events of the `ActionEvent` type:

```
class ButtonClickHandler implements EventHandler<ActionEvent>
{
   @Override
   public void handle(ActionEvent event)
   {
      // Write event handling code here.
   }
}
```

Notice the first line of the class declaration ends with the clause `implementsEventHandler <Actionevent>`. This specifies that the class implements the `EventHandler` interface, and the type of event object that it will work with is `ActionEvent`. Next, notice that the `handle` method has an `ActionEvent` parameter named `event`. When the `handle` method is called, an `ActionEvent` object will be passed to it as an argument. Any code you want to execute as the result of a `Button` click must be written in the `handle` method.

Once you have written an event handler class, you create an object of that class, and connect the event handler object with a control. When a `Button` control is clicked, it generates an `ActionEvent` object, and it automatically executes the `handle` method of the event handler object to which it is connected, passing the `ActionEvent` object as an argument. This is illustrated in Figure 12-30.

**Figure 12-30** A `Button` control firing an `ActionEvent`



```
public void handle(ActionEvent event)
{
   // Event handling code is here...
}
```

## Registering an Event Handler

Once you have written an event handler class, you create an object of the class, and connect the object to a control. The process of connecting an event handler object to a control is called *registering* the event handler. `Button` controls have a method named `setOnAction`, which is used for registering event handlers. You simply call the `Button`'s `setOnAction` method, passing a reference to the event handler object as an argument. This will connect the `Button` control to the event handler object.

For example, suppose we create a `Button` control named `myButton`, and we have written an event handler class named `ButtonClickHandler`. The following code registers an object of the `ButtonClickHandler` class with the `Button` control:

```
myButton.setOnAction(new ButtonClickHandler());
```

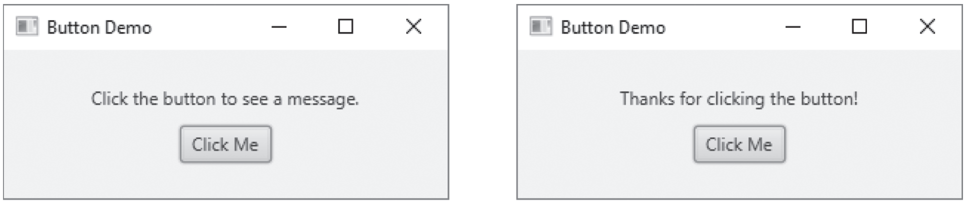
Now, when the user clicks on the `Button` control, the `ButtonClickHandler` object's `handle` method will be automatically executed.

A common technique for writing an event listener class is to write it as a private inner class inside the class that creates the GUI. Code Listing 12-11 shows an example. The program's output is shown in Figure 12-31. The image on the left shows the application when it starts, and the image on the right shows the application after the user has clicked the `Button`.

**Code Listing 12-11** (`EventDemo.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Pos;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
  An Event Demo
*/

public class EventDemo extends Application
{
   // Field for the Label control
   private Label myLabel;

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label control.
      myLabel = new Label("Click the button to see a message.");

      // Create a Button control.
      Button myButton = new Button("Click Me");

      // Register the event handler.
      myButton.setOnAction(new ButtonClickHandler());

      // Put the Label and Button in a VBox with 10 pixels of spacing.
      VBox vbox = new VBox(10, myLabel, myButton);

      // Create a Scene with the VBox as its root node.
      Scene scene = new Scene(vbox, 300, 100);

      // Set the scene's alignment to center.
      vbox.setAlignment(Pos.CENTER);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Set the stage title.
      primaryStage.setTitle("Button Demo");

      // Show the window.
      primaryStage.show();
   }

   /**
      Event handler class for myButton
   */

   class ButtonClickHandler implements EventHandler<actionevent>
   {
      @Override
      public void handle(ActionEvent event)
      {
         myLabel.setText("Thanks for clicking the button!");
      }
   }
}
```

**Figure 12-31** The EventDemo application (Oracle Corporate Counsel)



Click the button to see a message.

Thanks for clicking the button!

Before the user clicks the `Button` After the user has clicked the `Button`

This application has a `Label` control, named `myLabel`, that initially displays *Click the button to see a message*. When the user clicks the `Button` control, the text that is displayed by `myLabel` changes to *Thanks for clicking the button!* Let's take a closer look at the program to see how it works:

- Line 8 imports the `javafx.event.EventHandler` class, and line 9 imports the `javafx.event.ActionEvent` class. These classes are needed by our event handler class.
- Line 18 declares `myLabel` as a field in the `EventDemo` class. The reason we are declaring it as a field (instead of declaring it as a local variable in the `start` method) is that the event handler object needs to access it.
- Inside the `start` method, in line 30, we create a `Label` control and assign it to the `myLabel` field.
- In line 33, we create a `Button` control named `myButton`.
- Line 36 calls the `myButton` control's `setOnAction` method to register an event handler with the `Button`. To accomplish this, the code in this line does two things: it creates an instance of the `ButtonClickHandler` class (which we will see in a moment), and it passes a reference to that object as an argument to the `setOnAction` method.
- Lines 39 through 54 perform all the necessary steps to place the `Label` and the `Button` in a `VBox`, add the `VBox` to a scene, and add the scene to the stage. A title is displayed in the window, and the window is displayed.
- The event handler appears in lines 61 through 68. It is an inner class named `ButtonClickHandler`. The `handle` method appears in lines 64 through 67. Notice the `handle` method has one parameter, `event`, which is an `ActionEvent` object. This parameter receives the event object that is passed to the method when it is called. Although we do not actually use the `event` parameter in this method, we still have to list it inside the method header's parentheses, because it is required by the `ActionListener` interface.
- The statement in line 66 calls the `myLabel` object's `setText` method to change the text that the `Label` displays.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

12.29 What is an event? Give an example.

12.30 What is an event handler?

12.31 In what package is the `EventHandler` interface?

12.32 What method do you call to register an event handler with a `Button` control?

## 12.7 Reading Input with `TextField` Controls

**CONCEPT: The `TextField` control is a rectangular area that can accept keyboard input from the user.**

VideoNote The TextField Control

One of the primary controls that you will use to get data from the user is the `TextField` control. A `TextField` control appears as a rectangular area. When the application is running, the user can type text into a `TextField` control. In the program, you can retrieve the text that the user entered and use that text in any necessary operations.

To create a `TextField` control, you will use the `TextField` class, which is in the `javafx.scene.control` package. Once you have written the necessary `import` statement, you will create an instance of the `TextField` class. The following statement shows an example:

```
TextField myTextField = new TextField();
```

This statement creates an empty `TextField`. Optionally, you can pass a string to the constructor to display initial text in the `TextField`. Here is an example:

```
TextField myTextField = new TextField("Example data");
```

To retrieve the text that the user has typed into a `TextField` control, you call the control's `getText` method. The method returns the value that has been entered into the `TextField` as a `String`. For example, assume `myTextField` is a `TextField` control. The following code reads the value that has been entered into `myTextField`, and assigns it to a `String` variable named `input`:

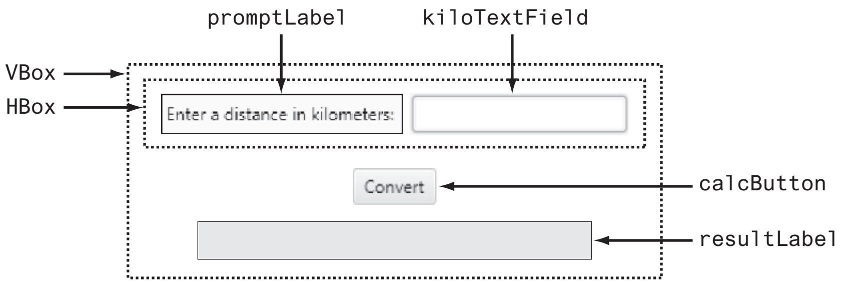
```
String input;
input = myTextField.getText();
```

To demonstrate the `TextField` control, we will look at the Kilometer Converter application in Code Listing 12-12. This application presents a window in which the user can enter a distance in kilometers, then click a button to see that distance converted to miles. The conversion formula is:

$$Miles = Kilometers \times 0.6214$$

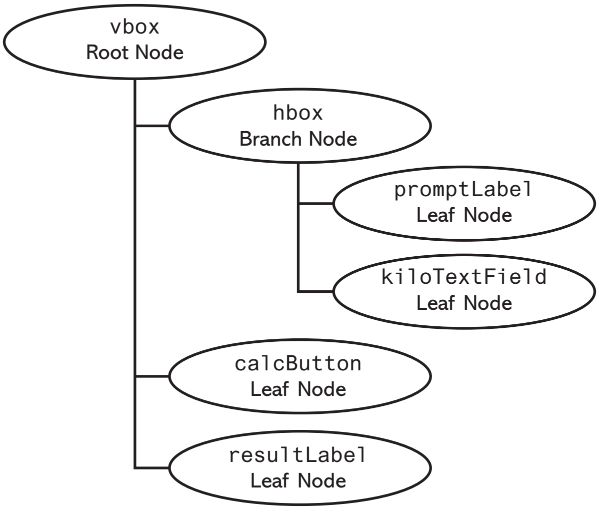
Figure 12-32 shows the layout of the controls in the Kilometer Converter application. Notice we use two layout containers: an `HBox` nested inside of a `VBox`. The `HBox` contains the `promptLabel` control and the `kiloTextField` control. The `VBox`, which is the root node,

**Figure 12-32** Layout of the controls in the Kilometer Converter application (Oracle Corporate Counsel)



contains the `HBox`, the `calcButton` control, and the `resultLabel` control. Figure 12-33 depicts the scene graph for the application.

**Figure 12-33** Scene graph for the Kilometer Converter application


vbox
Root Node
hbox
Branch Node
promptLabel
Leaf Node
kiloTextField
Leaf Node
calcButton
Leaf Node
resultLabel
Leaf Node

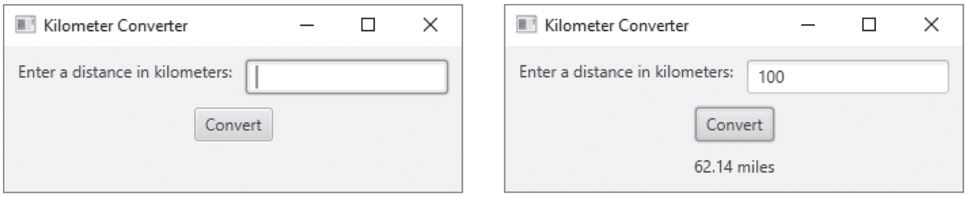
Code Listing 12-12 shows the application's code. The program's output is shown in Figure 12-34. The image on the left shows the application when it starts, and the image on the right shows the application after the user has entered 100 into the `kiloTextField` control, then clicked the `calcButton` control. Notice the conversion value is displayed in the `resultLabel` control.

**Code Listing 12-12** `(KiloConverter.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
  Kilometer Converter application
*/

```

```
public class KiloConverter extends Application
{
   // Fields
   private TextField kiloTextField;
   private Label resultLabel;

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label to display a prompt.
      Label promptLabel = new Label("Enter a distance in kilometers:");

      // Create a TextField for input.
      kiloTextField = new TextField();

      // Create a Button to perform the conversion.
      Button calcButton = new Button("Convert");

      // Register the event handler.
      calcButton.setOnAction(new CalcButtonHandler());

      // Create an empty Label to display the result.
      resultLabel = new Label();

      // Put the promptLabel and the kiloTextField in an HBox.
      HBox hbox = new HBox(10, promptLabel, kiloTextField);

      // Put the HBox, calcButton, and resultLabel in a VBox.
      VBox vbox = new VBox(10, hbox, calcButton, resultLabel);

      // Set the VBox's alignment to center.
      vbox.setAlignment(Pos.CENTER);

      // Set the VBox's padding to 10 pixels.
      vbox.setPadding(new Insets(10));

      // Create a Scene.
      Scene scene = new Scene(vbox);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

66        // Set the stage title.
67        primaryStage.setTitle("Kilometer Converter");
68
69        // Show the window.
70        primaryStage.show();
71     }
72
73     /**
74       Event handler class for calcButton
75     */
76
77     class CalcButtonHandler implements EventHandler<actionevent>
78     {
79        @Override
80        public void handle(ActionEvent event)
81        {
82           // Get the kilometers.
83           double kilometers = Double.parseDouble(kiloTextField.getText());
84
85           // Convert the kilometers to miles.
86           double miles = kilometers * 0.6214;
87
88           // Display the results.
89           resultLabel.setText(String.format("%,.2f miles", miles));
90        }
91     }
92  }
```

**Figure 12-34** The Kilometer Converter application (Oracle Corporate Counsel)


Kilometer Converter
Enter a distance in kilometers:
Convert
Kilometer Converter
Enter a distance in kilometers:
Convert

100

62.14 miles

When the application starts

After the user has entered *100* and clicked the `Button`

Let's take a closer look at the application:

- Lines 1 through 12 have the necessary JavaFX `import` statements.
- Lines 21 and 22 declare `kiloTextField` and `resultLabel` as fields in the `KiloConverter` class. The reason we are declaring them as fields (instead of declaring them as local variables in the `start` method) is that the event handler object needs to access them.
- Line 34 creates a `Label` control named `promptLabel`, to prompt the user to enter a distance in kilometers.

- Line 37 creates a `TextField` control and assigns it to the `kiloTextField` field.
- Line 40 creates a `Button` control named `calcButton`.
- Line 43 calls the `calcButton` control's `setOnAction` method to register an event handler with the `Button`. To accomplish this, the code in this line does two things: it creates an instance of the `CalcButtonHandler` class (which we will see in a moment), and it passes a reference to that object as an argument to the `setOnAction` method.
- Line 46 creates a `Label` control and assigns it to the `resultLabel` field. Notice we did not pass a string argument to the `Label` class constructor. As a result, the `Label` will be empty (it will not initially display any text).
- Line 49 puts the `promptLabel` and `kiloTextField` controls in an `HBox`, with 10 pixels of spacing between them.
- Line 52 puts the `HBox`, the `calcButton` control, and the `resultLabel` control in a `VBox`, with 10 pixels of spacing between them.
- Lines 55 and 58 set the `VBox`'s alignment to center, and padding to 10 pixels.
- Lines 61 through 70 perform all the necessary steps to add the `VBox` to a scene, and add the scene to the stage. A title is displayed in the window, and the window is displayed.
- The event handler appears in lines 77 through 91. It is an inner class named `CalcButtonHandler`. The `handle` method appears in lines 80 through 90. The statement in line 83 gets the value that the user entered into the `kiloTextField` control, converts it to a `Double`, and assigns it to the `kilometers` variable. Line 86 converts the kilometers to miles and assigns the result to the `miles` variable. Line 89 displays the results in the `resultLabel` control.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.33 In what package is the `TextField` class?

12.34 How do you retrieve the text that the user has entered into a `TextField` control?

## 12.8 Using Anonymous Inner Classes and Lambda Expressions to Handle Events

**CONCEPT: Event handler code can sometimes be simplified with anonymous inner classes or lambda expressions.**

### Using Anonymous Inner Classes to Create Event Handlers

VideoNote Using Anonymous Inner Classes as Event Handlers

If you look back at Code Listing 12-11 and Code Listing 12-12, you will notice that each program's event handler classes are instantiated only once: when the event handler is registered with a control. Recall from Chapter 10 that when a class is instantiated only once in a program, you can use an anonymous inner class instead of a regular class definition. You use the `new` operator to simultaneously define an anonymous inner class, and create an instance of it. Code Listing 12-13 shows an example of how we can modify the Kilometer Converter application to use an anonymous inner class as the `calcButton` control's event handler.

**Code Listing 12-13** `(AnonInnerClassKiloConverter.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
   Kilometer Converter application that uses an
   anonymous inner class to create an event handler.
*/

public class AnonInnerClassKiloConverter extends Application
{
   // Fields
   private TextField kiloTextField;
   private Label resultLabel;

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label to display a prompt.
      Label promptLabel = new Label("Enter a distance in kilometers:");

      // Create a TextField for input.
      kiloTextField = new TextField();

      // Create a Button to perform the conversion.
      Button calcButton = new Button("Convert");

      // Create an event handler.
      calcButton.setOnAction(new EventHandler<actionevent>()
      {
         @Override
47             public void handle(ActionEvent event)
48             {
49                // Get the kilometers.
50                Double kilometers = Double.parseDouble(kiloTextField.getText());
51 
52                // Convert the kilometers to miles.
53                Double miles = kilometers * 0.6214;
54 
55                // Display the results.
56                resultLabel.setText(String.format("%,.2f miles", miles));
57             }
58          });
59 
60          // Create an empty Label to display the result.
61          resultLabel = new Label();
62 
63          // Put the promptLabel and the kiloTextField in an HBox.
64          HBox hbox = new HBox(10, promptLabel, kiloTextField);
65 
66          // Put the HBox, calcButton, and resultLabel in a VBox.
67          VBox vbox = new VBox(10, hbox, calcButton, resultLabel);
68 
69          // Set the VBox's alignment to center.
70          vbox.setAlignment(Pos.CENTER);
71 
72          // Set the VBox's padding to 10 pixels.
73          vbox.setPadding(new Insets(10));
74 
75          // Create a Scene.
76          Scene scene = new Scene(vbox);
77 
78          // Add the Scene to the Stage.
79          primaryStage.setScene(scene);
80 
81          // Set the stage title.
82          primaryStage.setTitle("Kilometer Converter");
83 
84          // Show the window.
85          primaryStage.show();
86       }
87    }
```

In this version of the program, we have eliminated the definition of the `CalcButtonHandler` class. Instead, in lines 44 through 58, we instantiate an anonymous inner class that implements the `EventHandler` interface, and we register that object as the `calcButton` control's event handler.

## Using Lambda Expressions to Create Event Handlers

VideoNote
Using Lambda Expressions as Event Handlers

Recall our discussion of functional interfaces and lambda expressions in Chapter 10. A functional interface is an interface that has one, and only one, abstract method. The `EventHandler` interface has only one abstract method (the `handle` method), so it is a functional interface. Any time you are writing Java code to instantiate an anonymous class that implements a functional interface, you should consider using a lambda expression instead. A lambda expression is more concise than the code for instantiating an anonymous class. Code Listing 12-14 shows how we can use a lambda expression to further simplify the Kilometer Converter application. The lambda expression appears in lines 43 through 53. If you compare Code Listing 12-14 with the program shown in Code Listing 12-13, you can see that the lambda expression is more concise than the anonymous inner class declaration.

**Code Listing 12-14** `(LambdaKiloConverter.java)`

```
 1  import javafx.application.Application;
 2  import javafx.stage.Stage;
 3  import javafx.scene.Scene;
 4  import javafx.scene.layout.HBox;
 5  import javafx.scene.layout.VBox;
 6  import javafx.geometry.Pos;
 7  import javafx.geometry.Insets;
 8  import javafx.scene.control.Label;
 9  import javafx.scene.control.TextField;
10  import javafx.scene.control.Button;
11  import javafx.event.EventHandler;
12  import javafx.event.ActionEvent;
13
14  /**
15    Lambda version of the Kilometer Converter application
16  */
17
18  public class LambdaKiloConverter extends Application
19  {
20     // Fields
21     private TextField kiloTextField;
22     private Label resultLabel;
23
24     public static void main(String[] args)
25     {
26        // Launch the application.
27        launch(args);
28     }
29
30     @Override
31     public void start(Stage primaryStage)
32     {
33        // Create a Label to display a prompt.
```

```
      Label promptLabel = new Label("Enter a distance in kilometers:");

      // Create a TextField for input.
      kiloTextField = new TextField();

      // Create a Button to perform the conversion.
      Button calcButton = new Button("Convert");

      // Create an event handler.
      calcButton.setOnAction(event ->
      {
         // Get the kilometers.
         double kilometers = Double.parseDouble(kiloTextField.getText());

         // Convert the kilometers to miles.
         double miles = kilometers * 0.6214;

         // Display the results.
         resultLabel.setText(String.format("%,.2f miles", miles));
      });

      // Create an empty Label to display the result.
      resultLabel = new Label();

      // Put the promptLabel and the kiloTextField in an HBox.
      HBox hbox = new HBox(10, promptLabel, kiloTextField);

      // Put the HBox, calcButton, and resultLabel in a VBox.
      VBox vbox = new VBox(10, hbox, calcButton, resultLabel);

      // Set the VBox's alignment to center.
      vbox.setAlignment(Pos.CENTER);

      // Set the VBox's padding to 10 pixels.
      vbox.setPadding(new Insets(10));

      // Create a Scene.
      Scene scene = new Scene(vbox);

      // Add the Scene to the Stage.
      primaryStage.setScene(scene);

      // Set the stage title.
      primaryStage.setTitle("Kilometer Converter");

      // Show the window.
      primaryStage.show();
   }
}
```

Although lambda expressions simplify the use of anonymous inner classes, they have limitations. Recall from Chapter 10 that a lambda expression can access only variables that are effectively `final`. Also, lambda expressions can become difficult to read if they contain a lot of code.

There are many different techniques for creating event handlers. In this chapter, you have learned three techniques:

- Writing the definition of an inner class that implements the `EventHandler` interface. Then, instantiating that class, and registering it with a control. You saw this technique in Code Listings 12-11 and 12-12.
- Instantiating an anonymous inner class that implements the `EventHandler` interface, and registering the object with a control. You saw this technique in Code Listing 12-13.
- Using a lambda expression to instantiate an anonymous inner class that implements the `EventHandler` interface, and registering the object with a control. You saw this technique in Code Listing 12-11.

Each technique has its own advantages. For example, technique #1, defining an inner class, is advantageous for writing event handlers with a lot of code. Technique #2, instantiating an anonymous inner class, is advantageous for writing event handlers that have only a small amount of code. Technique #3, using a lambda expression, is the best approach for writing event handlers that have a small amount of code, as long as the code accesses variables that are effectively `final`.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.35 Describe three different techniques discussed in this chapter for writing event handlers.

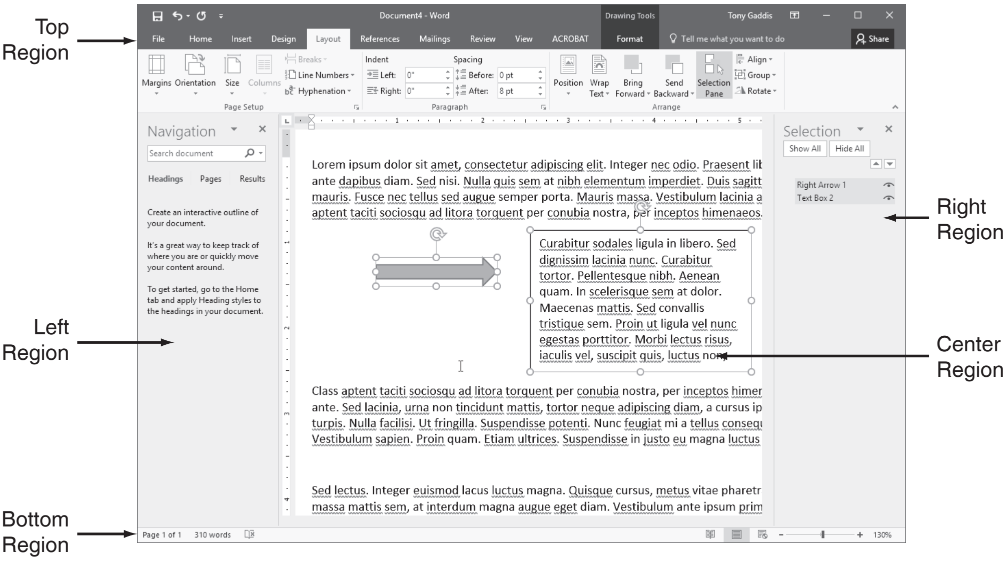
## 12.9 The `BorderPane` Layout Container

**CONCEPT: The `BorderPane` container displays its content in five regions. The regions are known as top, bottom, left, right, and center.**

Desktop applications commonly display a GUI with its controls organized in regions. For example, a GUI might display its main content in the center region of the window, while also displaying controls along the window's top, bottom, left, and right regions. Figure 12-35 shows an example using Microsoft Word. The document is displayed in the center of the window, while horizontal menu bars appear along the top, a status bar appears along the bottom, and vertically-arranged elements appear along the left and right edges of the window.

JavaFX provides a layout container named `BorderPane` that makes it easy to achieve this style of organization. The `BorderPane` layout container is divided into five regions: top, bottom, left, right, and center. The arrangement of these regions is shown in Figure 12-36. When a node is placed into a `BorderPane` layout container, it must be placed into one of these five regions.

**Figure 12-35** Typical GUI organization (Microsoft Corporation)


Top Region
Right Region
Left Region
Center Region
Bottom Region
Document4 - Word
Drawing Tools
Tony Gaddis
File
Home
Insert
Design
Layout
References
Mailings
Review
View
ACROBAT
Format
Tell me what you want to do
Share
Margins
Orientation
Size
Columns
Breaks
Line Numbers
Hyphenation
Page Setup
Indent
Left:
Right:
Spacing
Before:
After:
Paragraph
Position
Wrap Text
Bring Forward
Send Backward
Selection Pane
Align
Group
Rotate
Arrange
Navigation
Search document
Headings
Pages
Results
Selection
Show All
Hide All
Right Arrow 1
Text Box 2
Page 1 of 1
310 words
130%

Indent Left: 0" Right: 0"

Spacing Before: 0 pt After: 8 pt

Create an interactive outline of your document.

It's a great way to keep track of where you are or quickly move your content around.

To get started, go to the Home tab and apply Heading styles to the headings in your document.

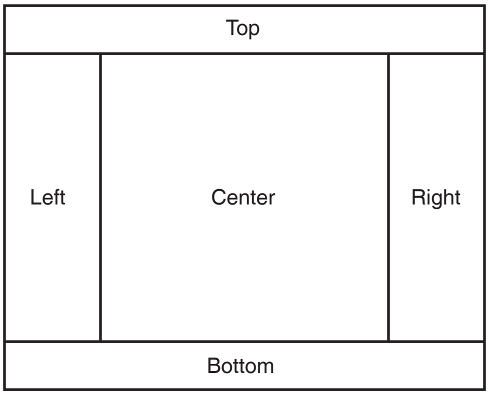
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec odio. Praesent lib
ante dapibus diam. Sed nisi. Nulla quis sem at nibh elementum imperdiet. Duis sagitt
mauris. Fusce nec tellus sed augue semper porta. Mauris massa. Vestibulum lacinia a
aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

Curabitur sodales ligula in libero. Sed dignissim lacinia nunc. Curabitur tortor. Pellentesque nibh. Aenean quam. In scelerisque sem at dolor. Maecenas mattis. Sed convallis tristique sem. Proin ut ligula vel nunc egestas porttitor. Morbi lectus risus, iaculis vel, suscipit quis, luctus non,

Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himer
ante. Sed lacinia, urna non tincidunt mattis, tortor neque adipiscing diam, a cursus ip
turpis. Nulla facilisi. Ut fringilla. Suspendisse potenti. Nunc feugiat mi a tellus consequ
Vestibulum sapien. Proin quam. Etiam ultrices. Suspendisse in justo eu magna luctus

Sed lectus. Integer euismod lacus luctus magna. Quisque cursus, metus vitae pharetr
massa mattis sem, at interdum magna augue eget diam. Vestibulum ante ipsum prim

**Figure 12-36** The regions of a BorderPane layout container


Top
Left
Center
Right
Bottom

Only one object at a time may be placed into a `BorderPane` region. For this reason, you do not usually put controls directly into a `BorderPane` region. Typically, you put controls into another type of layout container (such as an `HBox` or a `VBox`), then you put that container into one of the `BorderPane` regions.

**NOTE:** You don't have to use all of a `BorderPane`'s regions. If a `BorderPane` region doesn't contain anything, that region simply doesn't appear in the GUI.

To create a `BorderPane`, you import the `BorderPane` class from the `javafx.scene.layout` package. Then, you create an instance of the `BorderPane` class using one of three constructors described in Table 12-3.

**Table 12-3** BorderPane constructors

| Constructor | Description |
|---|---|
| `BorderPane()` | The no-arg constructor creates an empty `BorderPane` container. |
| `BorderPane(`*center*`)` | This constructor accepts one argument. The node that is passed as the argument is placed in the `BorderPane`'s center region. |
| `BorderPane(`*center*`,` *top*`,` *right*`,` *bottom*`,` *left*`)` | This constructor accepts five nodes as arguments: one to place in each region. |

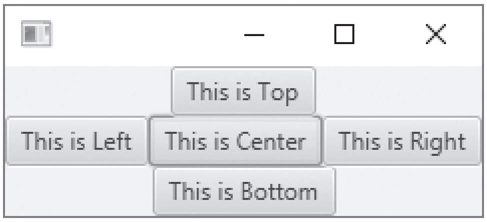
If you use the first constructor (the no-arg constructor), you will have to use the following `BorderPane` methods to put nodes in the container's regions: `setCenter`, `setTop`, `setBottom`, `setLeft`, and `setRight`. Code Listing 12-15 demonstrates these methods. The program's output is shown in Figure 12-37.

**Code Listing 12-15** (BorderPaneDemo1.java)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.layout.BorderPane;
import javafx.geometry.Pos;

/**
  A BorderPane Demo
*/

public class BorderPaneDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create some buttons.
      Button centerButton = new Button("This is Center");
```

```
        Button topButton = new Button("This is Top");
        Button bottomButton = new Button("This is Bottom");
        Button leftButton = new Button("This is Left");
        Button rightButton = new Button("This is Right");

        // Add each button to its own layout container.
        HBox centerHBox = new HBox(centerButton);
        HBox topHBox = new HBox(topButton);
        HBox bottomHBox = new HBox(bottomButton);
        VBox leftVBox = new VBox(leftButton);
        VBox rightVBox = new VBox(rightButton);

        // Set the alignment for the top and bottom.
        topHBox.setAlignment(Pos.CENTER);
        bottomHBox.setAlignment(Pos.CENTER);

        // Create a BorderPane.
        BorderPane borderPane = new BorderPane();

        // Add the buttons to the BorderPane's regions.
        borderPane.setCenter(centerHBox);
        borderPane.setTop(topHBox);
        borderPane.setBottom(bottomHBox);
        borderPane.setLeft(leftVBox);
        borderPane.setRight(rightVBox);

        // Create a Scene with the BorderPane as its root node.
        Scene scene = new Scene(borderPane);

        // Add the Scene to the Stage.
        primaryStage.setScene(scene);

        // Show the window.
        primaryStage.show();
    }
}
```

**Figure 12-37** The BorderPaneDemo1 GUI (Oracle Corporate Counsel)



Let's take a closer look at the code:

- Lines 26 through 30 create five button controls.
- Lines 33 through 37 put those button controls into their own layout containers. The `centerButton`, `topButton`, and `bottomButton` controls are added to `HBox` containers, and the `leftButton` and `rightButton` controls are added to `VBox` containers.
- Line 44 creates a `BorderPane` container.
- Lines 47 through 51 add the `HBox` and `VBox` containers to the regions of the `BorderPane`.

The second `BorderPane` constructor shown in Table 12-3 accepts one argument: the node to place in the center region. To place nodes in the other regions, you will use the following `BorderPane` methods: `setTop`, `setBottom`, `setLeft`, and `setRight`. The following code snippet shows lines 43 through 50 of the program `BorderPaneDemo2.java`. The program's output is the same as that shown in Figure 12-37.

```
43  // Create a BorderPane with a node in the center.
44  BorderPane borderPane = new BorderPane(centerHBox);
45
46  // Add the buttons to the BorderPane's regions.
47  borderPane.setTop(topHBox);
48  borderPane.setBottom(bottomHBox);
49  borderPane.setLeft(leftVBox);
50  borderPane.setRight(rightVBox);
```

The third constructor shown in Table 12-3 accepts five nodes as arguments: one to place in each region. This is the constructor you will normally want to use when you are going to place nodes in every region, and those nodes exist when the `BorderPane` is created. The following code snippet shows lines 43 through 46 of the program BorderPaneDemo3.java. The program's output is the same as that shown in Figure 12-37.

```
43  // Create a BorderPane.
44  BorderPane borderPane =
45   new BorderPane(centerHBox, topHBox, rightVBox,
46                  bottomHBox, leftVBox);
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

12.36 What are the names of the regions in a `BorderPane` container?

12.37 Describe the three `BorderPane` constructors.

## 12.10 The `ObservableList` Interface

**CONCEPT: Many of the containers in the JavaFX library implement the `ObservableList` interface. It is helpful to learn a few of the methods specified in the interface.**

As you spend more time developing JavaFX applications, you will frequently encounter the `ObservableList` interface. An object that implements the `ObservableList` interface is a special type of list that can fire an event handler any time an item in the list changes. Although many of the details of the `ObservableList` interface are beyond the scope of this text, you will commonly work with JavaFX containers that store nodes in an `ObservableList`. Learning a few basic `ObservableList` operations will give you more control over the JavaFX containers with which you will be working.

Table 12-4 lists a few of the `ObservableList` methods. Notice some of the methods are inherited from the `Collection` interface, of which is a `ObservableList` subinterface.

**Table 12-4** A few of the `ObservableList` methods

| Method | Description |
|---|---|
| `add(`*item*`)` | Adds a single item to the list. (This method is inherited from the `Collection` interface.) |
| `addAll(`*item*`...)` | Adds one or more items to the list, specified by the variable argument list. |
| `clear()` | Removes all of the items from the list. |
| `remove(`*item*`)` | Removes the object specified by *item* from the list. (This method is inherited from the `Collection` interface.) |
| `removeAll(`*item*`...)` | Removes one or more items to the list, specified by the variable argument list. |
| `setAll(`*item*`...)` | Clears the current contents of the list and adds all of the items specified by the variable argument list. |
| `size()` | Returns the number of items in the list. (This method is inherited from the `Collection` interface.) |

For example, layout containers keep their children nodes in an `ObservableList`. All layout containers have a method named `getChildren()` that returns their `ObservableList` of nodes. The following code snippet shows how you can create an empty `HBox` container, then use the `ObservableList`'s `addAll` method to add nodes to the `HBox`:

```
// Create three Label controls.
Label label1 = new Label("This is label1.");
Label label2 = new Label("This is label2.");
Label label3 = new Label("This is label3.");

// Create an empty HBox container.
HBox hbox = new HBox();

// Add the Label controls to the HBox.
hbox.getChildren().addAll(label1, label2, label3);
```

The following code shows how to remove `label1` from the HBox:

```
hbox.getChildren().remove(label1);
```

## 12.11 Common Errors to Avoid

- **Forgetting to extend the `Application` class in your JavaFX application.** The `Application` class in the `javafx.application` package must be extended by all JavaFX applications.
- **Forgetting to specify the `file:` protocol when loading an image file with the `Image` class.** When creating an instance of the `Image` class and passing a file name to the constructor, be sure to include the `file:` protocol along with the file name.
- **Trying to display an image using only the `Image` class.** The `Image` class loads an image file, but you also need the `ImageView` class to add the image to your scene graph.
- **Resizing an image but forgetting to preserve the aspect ratio.** If you resize an image without preserving its aspect ratio, the image can appear vertically or horizontally stretched. The `ImageView` class has a method named `setPreserveRatio` you can call to make sure that an image's aspect ratio is preserved.
- **Forgetting to register an event handler.** Even if you write an event handler class, it will not execute unless it has been registered with the correct control.

## Review Questions and Exercises

### Multiple Choice and True/False

1. The primary purpose of this type of control is to display text.
   a. `Button`
   b. `Label`
   c. `Message`
   d. `Text`
2. This type of control appears as a rectangular region that can accept keyboard input from the user.
   a. `Button`
   b. `Label`
   c. `TextField`
   d. `InputField`
3. Typically, when the user clicks this type of control, an action takes place.
   a. `Button`
   b. `Label`
   c. `TextField`
   d. `Switch`
4. In memory, the GUI objects in a scene are organized as nodes in this type of structure.
   a. scene array
   b. scene graph
   c. scene tree
   d. scene stack

5. There can be only one of these in a scene. It is the parent of all the other nodes.
   a. leaf node
   b. master node
   c. root node
   d. branch node
6. This type of node can have other nodes as children.
   a. leaf node
   b. master node
   c. anchor node
   d. branch node
7. This type of node cannot have other nodes as children.
   a. leaf node
   b. root node
   c. container node
   d. branch node
8. All JavaFX applications must extend the __________ class.
   a. `JavaFX`
   b. `Application`
   c. `GUI`
   d. `Window`
9. This container arranges its contents in a single, vertical row.
   a. `VerticalBox`
   b. `VBox`
   c. `GridPane`
   d. `HBox`
10. You use this class to load an image file into memory.
    a. `ImageView`
    b. `ImageLoader`
    c. `Image`
    d. `Img`
11. You use this class to actually display an image.
    a. `ImageView`
    b. `ImageLoader`
    c. `Image`
    d. `Img`
12. The `EventHandler` interface specifies a method named __________.
    a. `register`
    b. `onClick`
    c. `fire`
    d. `handle`
13. The `BorderPane` layout container has five regions known as which of the following?
    a. north, south, east, west, center
    b. top, bottom, left, right, center
    c. up, down, left, right, middle
    d. I, II, III, IV, and V

14. What type of object does a layout container's `getChildren()` method return?
    a. `ExpandableList`
    b. `MoveableList`
    c. `ArrayList`
    d. `ObservableList`
15. **True or False:** The `Stage` class is in the `javafx.scene` package.
16. **True or False:** The `HBox` container makes all of the nodes it contains of the same height.
17. **True or False:** You can nest a layout container inside of another layout container.
18. **True or False:** Only one object at a time may be placed into a `BorderPane` region.

## Find the Error

Find the error in each of the following code segments:

1.
```
// This code has an error!
@Override
public void start(Scene primaryScene)
{
   primaryScene.show();
}
```

2.
```
// This code has an error!
Label messageLabel = new Label("Hello World");
HBox hbox = new HBox();
hbox.add(messageLabel);
```

3. Assume `hbox` is an `HBox` container:
```
// This code has an error!
hbox.setAlignment(CENTER);
```

4.
```
// This code has an error!
ImageView view = new ImageView("file:HotAirBalloon.jpg");
```

5.
```
// This code has an error!
myImageView.setWidth(100);
myImageView.setHeight(100);
```

## Algorithm Workbench

1. Write a statement, after the following code, that creates an `HBox` container, and adds the `label1`, `label2`, and `label3` controls to it:
```
Label label1 = new Label("One");
Label label2 = new Label("Two");
Label label3 = new Label("Three");
```

2. Assume the `hbox` variable references an `HBox` container. Write a statement that creates a `Scene` object, adds `hbox` as the root node, and sets the size of the scene to 300 pixels wide by 200 pixels high.

3. Assume `primaryStage` references the `Stage` object, and `scene` references your `Scene` object. Write a statement that adds the scene to the stage.
4. Assume the `hbox` variable references an `HBox` container. Write a statement that center-aligns all controls in the `hbox` container.
5. Assume your JavaFX application is in the same directory as an image file named Cat.png. Write the code to create the `Image` and `ImageView` objects you will need to display the image.
6. Write a statement that creates an `Image` object, loading the Cat.png file from the following Internet location: http://www.greendale.edu/images/
7. Write a statement, after the following code, that creates an `HBox` container, adds the `label1`, `label2`, and `label3` controls to it, and has 10 pixels of spacing between the controls.

```
Label label1 = new Label("One");
Label label2 = new Label("Two");
Label label3 = new Label("Three");
```

8. Assume the `vbox` variable references a `VBox` container. Write a statement that adds 10 pixels of padding around the inside edge of the container.
9. Assume `gridPane` is the name of a `GridPane` container, and `button` is the name of a `Button` control. Write a statement that adds `button` to `gridPane` at column 2, row 5.
10. Write a statement that creates a `Button` control with the text *Click Me* displayed on the button.
11. Assume a JavaFX application has a `Button` control named `myButton`, and a `Label` control named `outputLabel`. Write an event handler class that can be used with the `Button` control. The event handler class should display the string “Hello World” in `outputLabel`.
12. In question 11, you wrote the code for an event handler class. Write a statement that registers an instance of the class with the `myButton` control.
13. In question 11, you wrote the code for an event handler class, and in question 12, you wrote the code to register an instance of that class with the `myButton` control. Rewrite the code as a lambda expression.
14. Assume `borderPane` is the name of an existing `BorderPane` object. Write statements that add the following existing components to the `BorderPane`:
    - Add `label1` to the top region
    - Add `label2` to the bottom region
    - Add `label3` to the center region
    - Add `label4` to the left region
    - Add `label5` to the right region

### Short Answer

1. What is an event-driven program?
2. What is the purpose of the `Application` class’s `launch` method?
3. What is the purpose of the `Application` class’s abstract `start` method?
4. What purpose do layout containers serve?

5. What package are the `Label`, `Button`, and `TextField` classes in?
6. What two classes do you use to display an image?
7. What is an image's aspect ratio? How do you make sure it is preserved when you resize an image?
8. Which layout container arranges controls in a horizontal row? Which layout container arranges controls in a vertical row? Which layout container arranges controls in rows and columns?
9. How do you retrieve the text that the user has entered into a `TextField` control?
10. Describe three different techniques discussed in this chapter for writing event handlers.
11. Which layout container divides a container into regions known as top, bottom, left, center, and right?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Latin Translator

VideoNote
The Latin Translator Problem

Look at the following list of Latin words and their meanings:

| Latin | English |
|---|---|
| sinister | left |
| dexter | right |
| medium | center |

Create a JavaFX application that translates the Latin words to English. The GUI should have three `Button`s, one for each Latin word. When the user clicks a `Button`, the application should display the English translation in a `Label`.

### 2. Name Formatter

Create a JavaFX application that lets the user enter the following pieces of data:

- The user's first name
- The user's middle name
- The user's last name
- The user's preferred title (Mr., Mrs., Ms., Dr., and so on.)

Assume the user has entered the following data:

- First name: *Kelly*
- Middle name: *Jane*
- Last name: *Smith*
- Title: *Ms.*

The application should have buttons that display the user's name formatted in the following ways:

*Ms. Kelly Jane Smith*
*Kelly Jane Smith*

*Kelly Smith*
*Smith, Kelly Jane, Ms.*
*Smith, Kelly Jane*
*Smith, Kelly*

### 3. Tip, Tax, and Total

Create a JavaFX application that lets the user enter the food charge for a meal at a restaurant. When a button is clicked, the application should calculate and display the amount of an 18 percent tip on the total food charge, 7 percent sales tax, and the total of all three amounts.

### 4. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. If an acre of land is valued at $10,000, its assessment value is $6,000. The property tax is then $0.64 for each $100 of the assessment value. The tax for the acre assessed at $6,000 will be $38.40. Create a GUI application that displays the assessment value and property tax when a user enters the actual value of a property.

### 5. Heads or Tails

Create a JavaFX application that simulates a coin being tossed. When the user clicks a button, the application should generate a random number in the range of 0 to 1. If the number is 0, the coin has landed on "heads," and if the number is 1, the coin has landed on "tails." Use an `ImageView` control, and the coin images in this book's Student Sample Programs to display the side of the coin when it is tossed.

### 6. Dice Simulator

Create a JavaFX application that simulates rolling a pair of dice. When the user clicks a button, the application should generate two random numbers, each in the range of 1 through 6, to represent the value of the dice. Use `ImageView` controls to display the dice. (In the Student Sample Programs, you will find six images named Die1.png, Die2.png, Die3.png, Die4.png, Die5.png, and Die6.png that you can use in the `ImageView` controls.)

### 7. Travel Expenses

Create a GUI application that calculates and displays the total travel expenses of a businessperson on a trip. Here is the information that the user must provide:

- Number of days on the trip
- Amount of airfare, if any
- Amount of car rental fees, if any
- Number of miles driven, if a private vehicle was used
- Amount of parking fees, if any
- Amount of taxi charges, if any
- Conference or seminar registration fees, if any
- Lodging charges, per night

The company reimburses travel expenses according to the following policy:

- $47 per day for meals
- Parking fees up to $20.00 per day

- Taxi charges up to $40.00 per day
- Lodging charges up to $195.00 per day
- If a private vehicle is used, $0.42 per mile driven

The application should calculate and display the following:

- Total expenses incurred by the businessperson
- The total allowable expenses for the trip
- The excess that must be paid by the businessperson, if any
- The amount saved by the businessperson if the expenses were under the total allowed

### 8. Joe's Automotive

Joe's Automotive performs the following routine maintenance services:

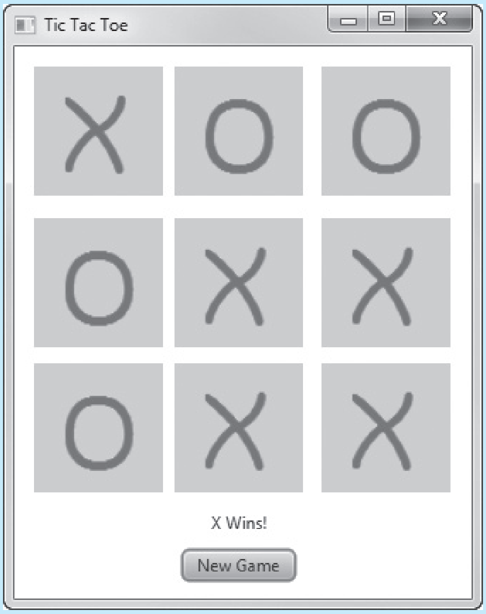
- Oil change: $35.00
- Lube job: $25.00
- Radiator flush: $50.00
- Transmission flush: $120.00
- Inspection: $35.00
- Muffler replacement: $200.00
- Tire rotation: $20.00

Joe also performs other nonroutine services and charges for parts and for labor ($60 per hour). Create a GUI application that displays the total for a customer's visit to Joe's.

### 9. Tic-Tac-Toe Simulator

Create a JavaFX application that simulates a game of tic-tac-toe. Figure 12-38 shows an example of the application's GUI. The GUI shown in the figure uses eight `ImageView` controls to display the Xs and Os. (You will find images for the X and the O in the book's Student Sample Files.)

**Figure 12-38** The tic-tac-toe application (Oracle Corporate Counsel)



X Wins!

The application should use a two-dimensional `int` array to simulate the game board in memory. When the user clicks the *New Game* button, the application should step through the array, storing a random number in the range of 0 through 1 in each element. The number 0 represents the letter O, and the number 1 represents the letter X. The application's window should then be updated to display the game board. The application should display a message indicating whether player X won, player Y won, or the game was a tie.

### 10. Slot Machine Simulation

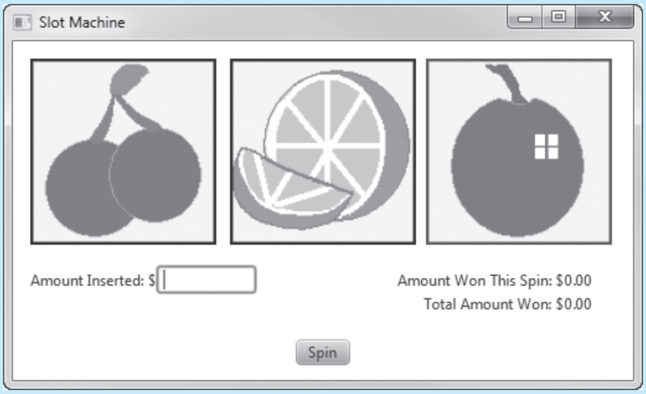
A slot machine is a gambling device into which the user inserts money then pulls a lever (or presses a button). The slot machine then displays a set of random images. If two or more of the images match, the user wins an amount of money that the slot machine dispenses back to the user.

Create a JavaFX application that simulates a slot machine. Figure 12-39 shows an example of how the GUI should look. The application should let the user enter into a `TextField` the amount of money he or she is inserting into the machine. When the user clicks the *Spin* button, the application should display three randomly selected symbols. (Slot machines traditionally display fruit symbols. You will find a set of fruit symbol images in the Student Sample Programs.) The program should also display the amount that the user won for the spin, and the total amount won for all spins.

The amount won for a spin is determined in the following way:

- If none of the randomly displayed images match, the user has won $0.
- If two of the images match, the user has won two times the amount entered.
- If three of the images match, the user has won three times the amount entered.

**Figure 12-39** Slot machine application (Oracle Corporate Counsel)



Amount Inserted: $

Amount Won This Spin: $0.00

Total Amount Won: $0.00

# CHAPTER 13 JavaFX: Advanced Controls

**TOPICS**

13.1 Styling JavaFX Applications with CSS
13.2 `RadioButton` Controls
13.3 `CheckBox` Controls
13.4 `ListView` Controls
13.5 `ComboBox` Controls
13.6 `Slider` Controls
13.7 `TextArea` Controls
13.8 Menus
13.9 The `FileChooser` Class
13.10 Using Console Output to Debug a GUI Application
13.11 Common Errors to Avoid

## 13.1 Styling JavaFX Applications with CSS

**CONCEPT:** **You can use Cascading Style Sheet rules to specify how a JavaFX application should appear.**

VideoNote
JavaFX and CSS

CSS, which stands for *Cascading Style Sheets*, is a simple language that specifies how a user interface should appear. CSS was originally created for Web designers. It gives designers a way to separate a webpage's content from its visual design. In other words, a webpage can use HTML to specify *what* information to display, and CSS to specify *how* that information should appear.

JavaFX supports CSS. When developing a JavaFX application, you can use CSS to specify many of the user interface's visual properties. For example, you can use CSS to specify font characteristics, background colors, background images, text alignment, border thickness, and much more.

When using CSS, you typically create one or more style sheets to accompany your JavaFX application. A *style sheet* is simply a text file that contains one or more *style definitions*, written in the following general format:

```
selector {
   property: value;
   property: value;
}
```

In the general format, *`selector`* is a name that determines the node or nodes that are affected by the style definition. Inside the braces, one or more *style rules* appear. A style rule takes the form

```
property:  value;
```

where *`property`* is the name of a property, and *`value`* is a value to assign to that property. Notice a colon separates the property name from the value, and the style rule ends with a semicolon. Here is an example of style definition:

```
.label {
  -fx-font-size: 20pt;
}
```

In this style definition, `.label` is the selector. Selector names that begin with a period are called *type selectors*, and they correspond to specific types of nodes in the JavaFX application. For example, the `.label` selector corresponds to `Label` controls. As a result, this style definition will be applied to all `Label` controls in the application. Inside the braces, we have one style rule that sets the `-fx-font-size` property to 20 points. So, this style definition specifies that all `Label` controls should display text in a 20-point font.

**NOTE:** In the previous example, notice the property name begins with `-fx-`. In JavaFX, all CSS properties begin with `-fx-`.

## Type Selector Names

You just learned that selector names beginning with a period are type selectors, and they correspond to specific types of JavaFX nodes. The type selector names are very similar to the JavaFX class names to which they correspond. Table 13-1 shows some examples. (Many of the classes shown in the table will be discussed later in this chapter.)

**Table 13-1** Some of the JavaFX nodes and their corresponding CSS type selectors

| Class Name in JavaFX | Corresponding Type Selector |
|---|---|
| `Button` | `.button` |
| `CheckBox` | `.check-box` |
| `CheckMenuItem` | `.check-menu-item` |
| `ComboBox` | `.combo-box` |
| `ImageView` | `.image-view` |
| `Label` | `.label` |
| `ListView` | `.list-view` |
| `Menu` | `.menu` |
| `MenuBar` | `.menu-bar` |
| `MenuItem` | `.menu-item` |
| `RadioButton` | `.radio-button` |
| `RadioMenuItem` | `.radio-menu-item` |
| `Slider` | `.slider` |
| `TextArea` | `.text-area` |
| `TextField` | `.text-field` |

As you can see from the table, there are some differences between the type selector names and the JavaFX class names. As previously mentioned, the type selectors begin with a period. After the period, the type selector names are written in all lowercase letters. When the JavaFX class name is composed of two words, the two words will be separated by a hyphen in the type selector name. So, the type selector for the `CheckBox` class is `.check-box`, and the type selector for the `TextField` class is `.text-field`.

## Style Properties

Each type of node in JavaFX has a set of style properties you can use. For example, Table 13-2 shows a few of the style properties for the `Labeled` class, and Table 13-3 shows a few properties for the `Region` class. The `Label` class, the `Button` class, the `CheckBox` class, and

**Table 13-2** A few CSS style properties for the `Labeled` class (inherited by the `Label` and `Button` classes)

| Property | Description | Possible Values |
|---|---|---|
| `-fx-text-alignment` | Specifies the control's text alignment | `left`<br>`center`<br>`right`<br>`justify` |
| `-fx-font-family` | Specifies the font for the control's text | Can be the name of a font installed on the local system, or one of the following generic fonts:<br>`serif`<br>`sans-serif`<br>`cursive`<br>`fantasy`<br>`monospace` |
| `-fx-font-size` | Specifies the size of control's text | Can be a number specified in points (`pt`) or pixels (`px`). Example:<br>`12pt`<br>`18px` |
| `-fx-font-style` | Specifies the style of control's text (normal, italic, or oblique) | `normal`<br>`italic`<br>`oblique` |
| `-fx-font-weight` | Specifies the weight of the control's text | You can specify one of these predefined values:<br>`normal`<br>`bold`<br>`bolder`<br>`lighter`<br>Alternatively, you can specify one of these numeric values: 100, 200, 300, 400, 500, 600, 700, 800, or 900. When using the numeric values, 400 is the same weight as `normal`, and 700 is the same weight as `bold`. |
| `-fx-text-fill` | Specifies the color of the control's text | Can be a hexadecimal color code, or a named color (see the discussion of color codes later in this section) |

**Table 13-3** A few CSS style properties for the `Region` class (inherited by the `Label` and `Button` classes)

| Property | Description | Possible Values |
|---|---|---|
| `-fx-border-style` | Specifies the type of border to draw around the control | There are many possible settings for this property, but the common ones are:<br>`none`<br>`solid`<br>`dotted`<br>`dashed` |
| `-fx-border-width` | Specifies the width of the control's border | Typically expressed as a number of pixels, such as `3px` |
| `-fx-border-color` | Specifies the color of the control's border | Can be a hexadecimal color code, or a named color (see the discussion of color codes later in this section) |
| `-fx-background-color` | Specifies the control's background color | Can be a hexadecimal color code, or a named color (see the discussion of color codes later in this section) |

the `RadioButton` class all inherit from the `Region` class and the `Labeled` class, so the properties in both of these tables may be used with `Label`, `Button`, `CheckBox`, and `RadioButton` controls. (The `CheckBox` and `RadioButton` controls will be discussed later in this chapter.)

Here is an example style definition that uses several of these properties:

```
.label {
   -fx-font-family: cursive;
   -fx-font-size: 14pt;
   -fx-font-style: italic;
   -fx-font-weight: bold;
   -fx-border-style: dotted;
}
```

If we use this style definition with a JavaFX application, its style rules will be applied to all `Label` controls in the user interface.

## Applying a Stylesheet to a JavaFX Application

As previously mentioned, a stylesheet is simply a text file that contains style definitions. Stylesheets are usually saved with the `.css` file extension. To apply a stylesheet to a JavaFX application, you save the stylesheet in the same directory as the JavaFX application. Then, in the application's code, you add the stylesheet to the scene. For example, suppose we have a stylesheet named *mystyles.css*, and in our Java code, the `scene` variable references our `Scene` object. We would use the following statement to add the stylesheet to the scene:

```
scene.getStylesheets().add("mystyles.css");
```

The `Scene` class's `getStylesheets()` method returns an object containing the scene's collection of stylesheets, and that object's `add` method adds the specified stylesheet to the collection.

Let's look at an example. Code Listing 13-1 shows the contents of the *demo1.css* file, and Code Listing 13-2 shows a JavaFX application that applies that stylesheet to its scene. The program's output is shown in Figure 13-1.

**Code Listing 13-1** (`demo1.css`)

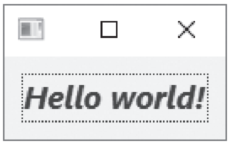
```
.label {
   -fx-text-alignment: center;
   -fx-font-family: cursive;
   -fx-font-size: 14pt;
   -fx-font-style: italic;
   -fx-font-weight: bold;
   -fx-border-style: dotted;
}
```

**Code Listing 13-2** (`CSSDemo1.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.geometry.Insets;
import javafx.geometry.Pos;

public class CSSDemo1 extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label control.
      Label myLabel = new Label("Hello world!");

      // Put the Label in a VBox.
      VBox vbox = new VBox(myLabel);
      vbox.setAlignment(Pos.CENTER);
      vbox.setPadding(new Insets(10));
```

```

        // Create a Scene and display it.
        Scene scene = new Scene(vbox);
        scene.getStylesheets().add("demo1.css");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

**Figure 13-1** Output of CSSDemo1.java (Oracle Corporate Counsel)



Hello world!

Let's look at another example. The stylesheet shown in Code Listing 13-3 has style definitions for `Label` and `Button` controls. The program in Code Listing 13-4 applies the stylesheet, and Figure 13-2 shows the program's output.

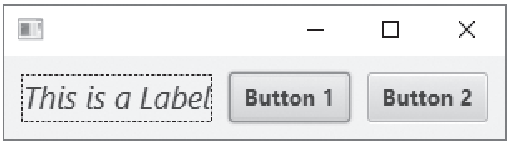
**Code Listing 13-3** (`demo2.css`)

```
.label {
   -fx-font-family: cursive;
   -fx-font-size: 14pt;
   -fx-font-style: italic;
   -fx-border-style: dashed;
}

.button {
   -fx-font-size: 10pt;
   -fx-font-weight: 900;
}
```

**Code Listing 13-4** (`CSSDemo2.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Insets;
import javafx.geometry.Pos;

public class CSSDemo2 extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label and some Buttons.
      Label label = new Label("This is a Label");
      Button button1 = new Button("Button 1");
      Button button2 = new Button("Button 2");

      // Put the controls in an HBox.
      HBox hbox = new HBox(10, label, button1, button2);
      hbox.setAlignment(Pos.CENTER);
      hbox.setPadding(new Insets(10));

      // Create a Scene and display it.
      Scene scene = new Scene(hbox);
      scene.getStylesheets().add("demo2.css");
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-2** Output of CSSDemo2.java (Oracle Corporate Counsel)



**TIP:** In a JavaFX application, CSS styling is applied at runtime. If you make any changes to the application's stylesheet, you do not have to recompile the Java application. The changes will take effect the next time you run the application.

**TIP:** JavaFX supports a large number of CSS properties (more than we can cover in this book). Part of the process of creating stylesheets is determining which properties are available for the nodes in your application. Oracle publishes a comprehensive CSS reference guide which documents all of the available properties. You can find the guide at:

`https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html`

## Applying Styles to the Root Node

If you want to apply styles to all of the nodes in a scene, use the `.root` selector. Code Listing 13-5 shows an example stylesheet with two style definitions: one for the root node, and another for `Label` controls. The program in Code Listing 13-6 applies the stylesheet, and Figure 13-3 shows the program's output. Notice both the `Label` and the `Button` display their text in a serif font. This is because it is specified by the `.root` style definition.

**Code Listing 13-5** `(demo3.css)`

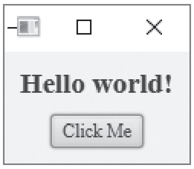
```
.root {
   -fx-font-family: serif;
}

.label {
   -fx-font-size: 14pt;
   -fx-font-weight: bold;
}
```

**Code Listing 13-6** `(CSSDemo3.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.geometry.Insets;
import javafx.geometry.Pos;

public class CSSDemo3 extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

```

```java
    @Override
    public void start(Stage primaryStage)
    {
        // Create a Label and a Button.
        Label myLabel = new Label("Hello world!");
        Button myButton = new Button("Click Me");

        // Put the controls in a VBox.
        VBox vbox = new VBox(10, myLabel, myButton);
        vbox.setAlignment(Pos.CENTER);
        vbox.setPadding(new Insets(10));

        // Create a Scene and display it.
        Scene scene = new Scene(vbox);
        scene.getStylesheets().add("demo2.css");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

**Figure 13-3** Output of CSSDemo3.java (Oracle Corporate Counsel)



If a style definition for a specific type of node contradicts a style rule in the `.root` definition, the more specific style definition takes precedence over the `.root` definition. To demonstrate, look at the style sheet in Code Listing 13-7. Notice the `.root` definition specifies a font size of 20 points, and the `.label` definition specifies a font size of 10 points. If an application uses this style sheet, its `Label` controls will display text at 10 points, and all other controls will display text at 20 points.

**Code Listing 13-7** (`demo4.css`)

```css
.root {
   -fx-font-size: 20pt;
}

.label {
   -fx-font-size: 10pt;
   -fx-font-weight: bold;
}
```

## Specifying Multiple Selectors in the Same Style Definition

You can list multiple selectors, separated by commas, in the first line of a style definition. Code Listing 13-8 shows an example. Notice line 1 lists the `.label` selector, followed by the `.button` selector. As a result, the style rules will be applied to `Label` controls and `Button` controls.

**Code Listing 13-8** `(demo5.css)`

```
1  .label, .button {
2     -fx-font-family: cursive;
3     -fx-font-size: 14pt;
4     -fx-font-style: italic;
5  }
```

## Working with Colors

CSS uses the *RGB* color system to define colors. In the RGB color system, all colors are created by mixing various shades of red, green, and blue. For example, if you mix bright red, bright green, and no blue you get yellow. If you mix bright red, bright blue, and no green, you get purple. White is created when you combine all three colors at their maximum brightness, and black is created when there is no red, green, or blue.

When you define a color in CSS, you specify an integer value in the range of 0 through 255 for each of the color components (red, green, and blue). The higher the number for a color component, the brighter that color component will be. The most common way to specify a color is with a hexadecimal number, prefixed with the # character. The general format of a hexadecimal color value is:

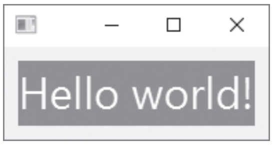
```
#RRGGBB
```

where `RR` is the hexadecimal value for red, `GG` is the hexadecimal value for green, and `BB` is the hexadecimal value for blue. Each of these color components must be a hexadecimal number in the range of `00` through `FF`. For example, the value `#05AAFF` specifies `05` for red, `AA` for green, and `FF` for blue. When these color components are combined, we get a light shade of blue.

To demonstrate, look at the style sheet in Code Listing 13-9. It defines a style for `Label` controls. The style definition specifies a font size of 20 points, a text fill color of `#FFFFFF` (white), and a background color of `#909090` (gray). Figure 13-4 shows an example of how a `Label` appears with this style applied.

**Code Listing 13-9** `(demo6.css)`

```
.label {
   -fx-font-size: 24pt;
   -fx-text-fill: #FFFFFF;
   -fx-background-color: #909090;
}
```



Hello world!

**Figure 13-4** Example `Label` with styles applied (Oracle Corporate Counsel)

If you aren't comfortable with hexadecimal values, you can use *functional RGB notation* to specify colors. The general format is:

```
rgb(red, green, blue)
```

In this notation, the values given for red, green, and blue can either be an integer in the range of 0 through 255, or a percentage in the range of 0 percent through 100 percent. For example, the notation `rgb(255, 0, 127)` specifies 255 for red, 0 for green, and 127 for blue. And, the notation `rgb(50%, 20%, 90%)` specifies that red should have 50 percent intensity, green should have 20 percent intensity, and blue should have 90 percent intensity.

To demonstrate, look at the style sheet in Code Listing 13-10. The `.label` style definition uses functional RGB notation to specify the same color values as previously shown in Code Listing 13-9. The style definition specifies the following:

- A font size of 24 points
- A text fill color where red = 255, green = 255, and blue = 255
- A background where red = 144, green = 144, and blue = 144

**Code Listing 13-10** (`demo7.css`)

```
.label {
   -fx-font-size: 24pt;
   -fx-text-fill: rgb(255, 255, 255);
   -fx-background-color: rgb(144, 144, 144);
}
```

### Named Colors

CSS provides numerous predefined names that represent colors. We refer to these names as the *named colors*. You can use these named colors just as you would use a hexadecimal color number, or a functional RGB expression. At the time of this writing, JavaFX supports 148 named colors. Table 13-4 lists only a few of them. You can see the entire list in Appendix K, which you can download from the computer science portal at www.pearsonhighered. com/gaddis. You can also see them in the official JavaFX CSS Reference Guide, available at:

```
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html
```

**Table 13-4** A few of the named colors

| | | | |
|---|---|---|---|
| `antiquewhite` | `aqua` | `black` | `blue` |
| `chocolate` | `crimson` | `darkgray` | `darkgreen` |
| `gold` | `gray` | `green` | `lavender` |
| `maroon` | `olive` | `orange` | `plum` |
| `purple` | `red` | `sandybrown` | `tan` |
| `teal` | `violet` | `white` | `yellow` |

Code Listing 13-11 shows an example style sheet that uses named colors. The `.label` style definition specifies white as the text fill color, and blue as the background color.

**Code Listing 13-11** (`demo8.css`)

```css
.label {
   -fx-text-fill: white;
   -fx-background-color: blue;
}
```

## Creating a Custom Style Class Name

In a type selector, the name that appears after the period is known as the *style class*. For example, in the `.label` selector, the style class is `label`, and in the `.button` selector, the style class is `button`. To achieve more flexibility in applying style definitions, JavaFX allows you to create your own custom style classes.

For example, suppose you want some of the `Button` controls in an application to appear with a black background and white text, and you want other `Button` controls to appear with a white background and black text. This is possible if you create two custom style classes: one for the black buttons, and another for the white buttons. Code Listing 13-12 shows an example style sheet in which we have created two custom style classes: one named `button-black`, and another named `button-white`.

**Code Listing 13-12** (`demo9.css`)

```css
.button-black {
   -fx-background-color: black;
   -fx-text-fill: white;
}

.button-white {
   -fx-background-color: white;
   -fx-text-fill: black;
}
```

In this style sheet, the `.button-black` style definition specifies a black background with white text. The `.button-white` style definition specifies a white background with black text. To apply either of these style definitions to a node, we must add the desired style class to the appropriate node in our Java application.

For example, suppose we have a `Button` control named `button1`, and we want to apply the `.button-black` style definition to it. We would use the following statement to add the `button-black` style class to `button1`:

```
button1.getStyleClass().add("button-black");
```

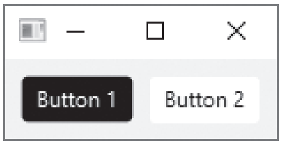
Each node keeps an internal collection of style classes. This collection determines which style definitions will be applied to the node. The `getStyleClass()` method returns an object containing the node's collection of style classes, and that object's `add` method adds the specified style class to the collection. (When you pass a style class name as an argument to the `add` method, you do not include the period.) Code Listing 13-13 shows a complete application that uses the custom style classes previously shown. The program's output is shown in Figure 13-5.

**Code Listing 13-13** (`CSSDemo9.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.Button;
import javafx.geometry.Insets;
import javafx.geometry.Pos;

public class CSSDemo9 extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create some Buttons.
      Button button1 = new Button("Button 1");
      button1.getStyleClass().add("button-black");

      Button button2 = new Button("Button 2");
      button2.getStyleClass().add("button-white");

      // Put the Buttons in an HBox.
      HBox hbox = new HBox(10, button1, button2);
      hbox.setAlignment(Pos.CENTER);
      hbox.setPadding(new Insets(10));
```

```

        // Create a Scene and display it.
        Scene scene = new Scene(hbox);
        scene.getStylesheets().add("demo9.css");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

**Figure 13-5** Output of CSSDemo9.java (Oracle Corporate Counsel)



## ID Selectors

Each node in the scene graph can optionally be assigned a unique identifier known as an *ID*. A node's ID is a string that can be used to identify the node in a style sheet. This is useful when you want to apply a style to one specific node.

You assign an ID to a node by calling the node's `setId` method. Here is an example:

```
Label outputLabel = new Label("File not found!");
outputLabel.setId("label-error");
```

In this example, `outputLabel` is a `Label` control, and we have assigned it the ID `"label-error"`. Next, we create a style definition with the selector `#label-error` in our style sheet. Here is an example:

```
#label-error {
    -fx-background-color: red;
    -fx-text-fill: white;
}
```

Notice the selector in this style definition begins with the # character. This indicates that it is an *ID selector*. The name that follows the # character is assumed to be the ID of a node. As a result, this style definition will be applied only to the node with this specific ID.

**NOTE:** All node IDs should be unique within a scene graph. Unfortunately, the Java compiler does not check for duplicate IDs, and no exception will be thrown at runtime if two nodes have the same ID. It is your responsibility, as the developer, to make sure that you do not assign the same ID to more than one node.

## Inline Style Rules

Style rules can be applied directly to a node in the application's Java code, using the node's `setStyle` method. You simply pass the style rule, as a string argument, to the method. Here is an example:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt");
```

This code creates a `Label` control named `outputLabel`, and sets the `Label`'s font size to 24 points. Notice the style rule argument does not end with a semicolon. In inline styling, semicolons are required only to separate multiple style rules. Here is an example that passes two style rules to the `setStyle` method:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt; -fx-font-weight: bold");
```

This code creates a `Label` control named `outputLabel`, sets the `Label`'s font size to 24 points, and sets the font's weight to bold.

Keep in mind the `setStyle` method removes any styles that were previously applied to the node, and replaces them with the new style rule that is passed as an argument. If you want to add a new style rule to a node's existing set of style rules, call the `getStyleClass().add()` method. The following code snippet shows an example:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt");
outputLabel.getStyleClass().add("-fx-font-weight: bold");
```

This code creates a `Label` control named `outputLabel`, sets the `Label`'s font size to 24 points, and sets the font's weight to bold.

Inline styling is convenient when you have only one or two style rules to apply in an application. In most cases, however, it is a better practice to use stylesheets. If you need to change a particular style rule, it will most likely be easier to find that rule in a style sheet, than in the application's Java code. Also, if you change a style rule that was applied inline, you will have to recompile the Java application. Changing a rule in a style sheet does not require that the application be recompiled.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.1 What is the type selector name that corresponds to each of the following JavaFX types?

```
Button
TextField
Label
ImageView
```

13.2 You haven't yet learned about the `MenuButton` control, but you have learned how to convert JavaFX class names to CSS type selectors. What would the type selector name for the `MenuButton` class be?

13.3 Which CSS style property specifies the font for a control's text?

13.4 If you want a control's text to appear in bold, which CSS style property would you set?

13.5 If you want to place a dotted border around a `Label` control, which CSS style property would you set?

13.6 Assume `scene` is the name of a `Scene` object, and stylesheet.css is the name of a CSS stylesheet. Write the statement that applies the stylesheet.css stylesheet to `scene`.

13.7 If you want to apply styles to all of the nodes in a scene, what selector would you use in a style definition?

13.8 If a style definition for a specific type of node contradicts a style rule in the `.root` definition, which takes precedence—the more specific style definition, or the `.root` definition?

13.9 When speaking of colors, what does RGB mean?

13.10 In the hexadecimal color value `#0511FF`, the `FF` value specifies which color component?

13.11 In a style definition, what type of selector begins with a # character?

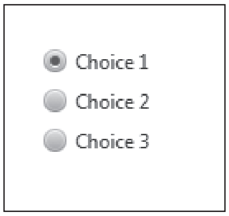
## 13.2 RadioButton Controls

**CONCEPT: `RadioButton` controls normally appear in groups of two or more and allow the user to select one of several possible options.**

VideoNote
`RadioButton` Controls

`RadioButton` controls are useful when you want the user to select one choice from several possible options. Figure 13-6 shows a group of `RadioButton` controls. A `RadioButton` may be selected or deselected. Each `RadioButton` has a small circle that appears filled-in when the `RadioButton` is selected, and appears empty when the `RadioButton` is deselected.

**Figure 13-6** `RadioButton` controls (Oracle Corporate Counsel)



● Choice 1
○ Choice 2
○ Choice 3

To create a `RadioButton` control, you will use the `RadioButton` class, which is in the `javafx.scene.control` package. Once you have written the necessary `import` statement, you will create an instance of the `RadioButton` class. The following statement shows an example of creating a `RadioButton` object that displays the text *Choice 1*:

```
RadioButton radio1 = new RadioButton("Choice 1");
```

Notice we pass the string we want displayed next to the `RadioButton` as an argument to the `RadioButton` class's constructor. Optionally, you can omit the string argument to create a `RadioButton` control with no text displayed next to it. Here is an example:

```
RadioButton radio1 = new RadioButton();
```

`RadioButton` controls are normally grouped together in a *toggle group*. Only one of the `RadioButton` controls in a toggle group may be selected at any time. Clicking on a `RadioButton` selects it and automatically deselects any other `RadioButton` in the same toggle group. Because only one `RadioButton` in a toggle group can be selected at any given time, the `RadioButton` controls are said to be *mutually exclusive*.

To create a toggle group, you use the `ToggleGroup` class, which is in the `javafx.scene.control` package. Here is an example of a statement that creates an instance of the class:

```
ToggleGroup myToggleGroup = new ToggleGroup();
```

After creating a `ToggleGroup` object, you must call each `RadioButton` control's `setToggleGroup` method to add them to the `ToggleGroup`. The following code snippet shows an example:

```
// Create some RadioButtons.
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
RadioButton radio3 = new RadioButton("Option 3");

// Create a ToggleGroup.
ToggleGroup radioGroup = new ToggleGroup();

// Add the RadioButtons to the ToggleGroup.
radio1.setToggleGroup(radioGroup);
radio2.setToggleGroup(radioGroup);
radio3.setToggleGroup(radioGroup);
```

## Determining in Code Whether a RadioButton Is Selected

To determine whether a `RadioButton` is selected, you call the `RadioButton` class's `isSelected` method. The method returns a `boolean` value indicating whether the `RadioButton` is selected. If the `RadioButton` is selected, the method returns `true`. Otherwise, it returns `false`. In the following code, the `radio1` variable references a `RadioButton`. The `if` statement calls the `isSelected` method to determine whether the `RadioButton` is selected.

```
if (radio1.isSelected())
{
    //  Code here executes if the radio
    //  button is selected.
}
```

## Selecting a RadioButton in Code

When you create a group of `RadioButton` controls, you usually want one of the `RadioButtons` to be initially selected. You can select a `RadioButton` in code with the `RadioButton` class's `setSelected` method. The `setSelected` method takes a `boolean` argument. If you pass `true` to the method, the `RadioButton` will be selected just as if the user had clicked it. (If you pass `false` as an argument to the `setSelected` method, the `RadioButton` will be deselected.)

In the following statement, the `radio1` variable references a `RadioButton`. When this statement executes, the `RadioButton` that is referenced by `radio1` will be selected. (Any other `RadioButton` in the same toggle group that is already selected will be deselected.)

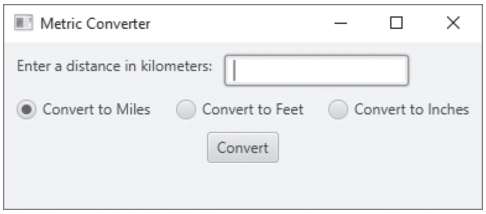
```
radio1.setSelected(true);
```

Let's look at an application that uses `RadioButton` controls. Code Listing 13-14 shows the code for the Metric Converter application, which converts kilometers to miles, feet, and inches. The conversion formulas are:

```
Miles  = Kilometers × 0.6214
Feet   = Kilometers × 3281.0
Inches = Kilometers × 39370.0
```

When the program runs, it displays the window shown in Figure 13-7. The user enters a distance in kilometers, then clicks a `RadioButton` to select miles, feet, or inches. When the user clicks the `Button` control, the program performs the selected conversion and displays the result. For example, Figure 13-8 shows three screenshots of the application. In the screens, the user has converted 100 kilometers to miles, feet, and inches.

**Figure 13-7** The Metric Converter application (Oracle Corporate Counsel)

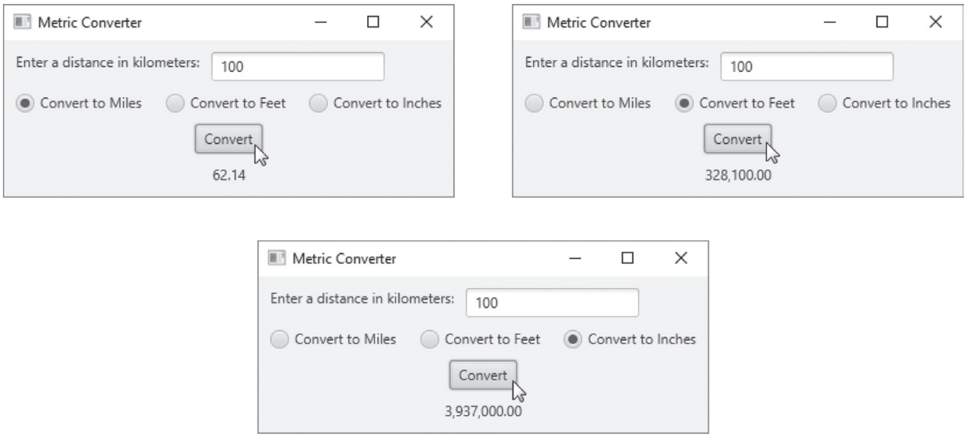


Metric Converter

Enter a distance in kilometers:

● Convert to Miles ○ Convert to Feet ○ Convert to Inches

**Figure 13-8** The Metric Converter application with conversions displayed (Oracle Corporate Counsel)



Metric Converter

Enter a distance in kilometers: 100

● Convert to Miles ○ Convert to Feet ○ Convert to Inches

62.14

Metric Converter

Enter a distance in kilometers: 100

○ Convert to Miles ● Convert to Feet ○ Convert to Inches

328,100.00

Metric Converter

Enter a distance in kilometers: 100

○ Convert to Miles ○ Convert to Feet ● Convert to Inches

3,937,000.00

**Code Listing 13-14** (`MetricConverter.java`)

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
  Metric Converter application
*/

public class MetricConverter extends Application
{
   // Fields
   private TextField kiloTextField;
   private Label resultLabel;
   private RadioButton milesButton;
   private RadioButton feetButton;
   private RadioButton inchesButton;

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label to display a prompt.
      Label promptLabel = new Label("Enter a distance in kilometers:");

      // Create a TextField for input.
      kiloTextField = new TextField();

      // Create the RadioButton controls.
      milesButton = new RadioButton("Convert to Miles");
      feetButton = new RadioButton("Convert to Feet");
      inchesButton = new RadioButton("Convert to Inches");
```

```

        // Select the milesButton control.
        milesButton.setSelected(true);

        // Add the RadioButton controls to a ToggleGroup.
        ToggleGroup radioGroup = new ToggleGroup();
        milesButton.setToggleGroup(radioGroup);
        feetButton.setToggleGroup(radioGroup);
        inchesButton.setToggleGroup(radioGroup);

        // Create a Button to perform the conversion.
        Button calcButton = new Button("Convert");

        // Register the event handler.
        calcButton.setOnAction(new CalcButtonHandler());

        // Create an empty Label to display the result.
        resultLabel = new Label();

        // Put the promptLabel and the kiloTextField in an HBox.
        HBox promptHBox = new HBox(10, promptLabel, kiloTextField);

        // Put the RadioButtons in an HBox.
        HBox radioHBox = new HBox(20, milesButton, feetButton,
                                  inchesButton);

        // Put everything in a VBox.
        VBox mainVBox = new VBox(10, promptHBox, radioHBox, calcButton,
                                 resultLabel);

        // Set the VBox's alignment to center.
        mainVBox.setAlignment(Pos.CENTER);

        // Set the VBox's padding to 10 pixels.
        mainVBox.setPadding(new Insets(10));

        // Create a Scene.
        Scene scene = new Scene(mainVBox);

        // Add the Scene to the Stage.
        primaryStage.setScene(scene);

        // Set the stage title.
        primaryStage.setTitle("Metric Converter");

```

```
 93         // Show the window.
 94         primaryStage.show();
 95      }
 96
 97    /**
 98      Event handler class for calcButton
 99    */
100
101    class CalcButtonHandler implements EventHandler<ActionEvent>
102    {
103       @Override
104       public void handle(ActionEvent event)
105       {
106          // Constants for the conversion factors.
107          final double MILES_CONVERSION = 0.6214;
108          final double FEET_CONVERSION = 3281.0;
109          final double INCHES_CONVERSION = 39370.0;
110
111          // Variable to hold the result
112          double result = 0;
113
114          // Get the kilometers.
115          double kilometers = Double.parseDouble(kiloTextField.getText());
116
117          // Perform the selected conversion.
118          if (milesButton.isSelected())
119          result = kilometers * MILES_CONVERSION;
120
121          if (feetButton.isSelected())
122          result = kilometers * FEET_CONVERSION;
123
124          if (inchesButton.isSelected())
125          result = kilometers * INCHES_CONVERSION;
126
127          // Display the results.
128          resultLabel.setText(String.format("%,.2f", result));
129       }
130    }
131 }
```

Let's take a closer look at the application:

- Lines 1 through 14 have the necessary JavaFX `import` statements.
- Lines 23 through 27 declare `kiloTextField`, `resultLabel milesButton`, `feetButton`, and `inchesButton` as fields in the `MetricConverter` class. The reason we are declaring them as fields (instead of declaring them as local variables in the `start` method) is because the event handler object needs to access them.

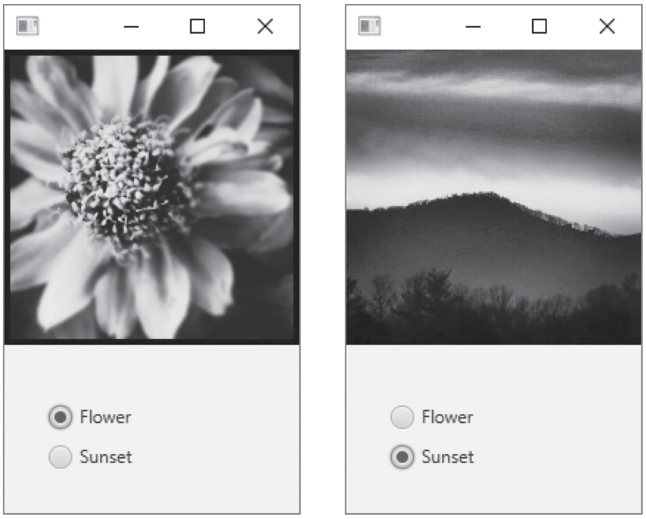
- Line 39 creates a `Label` control named `promptLabel`, to prompt the user to enter a distance in kilometers.
- Line 42 creates a `TextField` control and assigns it to the `kiloTextField` field.
- Lines 45 through 47 create the `RadioButton` controls and assign them to the `milesButton`, `feetButton`, and `inchesButton` fields.
- Line 50 causes the `milesButton` control to be initially selected.
- Line 53 creates a `ToggleGroup` object named `radioGroup`, and lines 54 through 56 add each of the `RadioButton` controls to the `ToggleGroup`.
- Line 59 creates a `Button` control named `calcButton`.
- Line 62 calls the `calcButton` control's `setOnAction` method to register an instance of the `CalcButtonHandler` class as an event handler with the `Button`.
- Line 65 creates an empty `Label` control and assigns it to the `resultLabel` field.
- Line 68 puts the `promptLabel` and `kiloTextField` controls in an `HBox` named `promptHBox`, with 10 pixels of spacing between them.
- Lines 71 and 72 put the `RadioButton` controls in an `HBox` named `radioHBox`, with 20 pixels of spacing between them.
- Lines 75 and 76 put the `promptHBox`, `radioHBox`, `calcButton`, and `resultLabel` controls in a `VBox`, with 10 pixels of spacing between them.
- Lines 79 and 82 set the `VBox`'s alignment to center, and the padding to 10 pixels.
- Lines 85 through 94 perform all the necessary steps to add the `VBox` to a scene, and add the scene to the stage. A title is displayed in the window, and the window is displayed.
- The event handler appears in lines 101 through 130. It is an inner class named `CalcButtonHandler`. The `handle` method appears in lines 104 through 129.
  - Lines 107 through 109 declare named constants for the necessary conversion factors.
  - Line 112 declares a local variable named `result`, to hold the result of the conversion.
  - Line 115 gets the value that the user entered into the `kiloTextField` control, converts it to a `Double`, and assigns it to the `kilometers` variable.
  - The `if` statements in lines 118 through 125 determine which `RadioButton` is selected, and performs the chosen conversion operation.
  - Line 128 displays the results in the `resultLabel` control.

## Responding to `RadioButton` Clicks

Sometimes, you merely want to know if a `RadioButton` is selected. When that is the case, you call the `RadioButton` control's `isSelected` method. Other times, however, you want an action to take place immediately when the user clicks a `RadioButton`. Just like `Button` controls, `RadioButton` controls generate an `ActionEvent` when they are clicked. If you want to respond to a `RadioButton`'s action event, you have to register an event handler with the control. The process of creating and registering an event handler for a `RadioButton` is the same as the process of creating and registering an event handler for a `Button`.

Code Listing 13-15 shows an example. When the program runs, it displays the window shown on the left in Figure 13-9. Initially the image of a flower is displayed. When the user clicks the *Sunset* `RadioButton`, the image immediately changes as shown in the window on the right. The user can click the `RadioButton` controls to display either the flower or the sunset.

**Figure 13-9** The RadioButtonEvent application



**Code Listing 13-15** (RadioButtonEvent.java)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
   A RadioButton ActionEvent Demo
*/

public class RadioButtonEvent extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create two Image objects.
      Image flowerImage = new Image("file:Flower.jpg");
      Image sunsetImage = new Image("file:Sunset.jpg");

      // Create an ImageView object.
      ImageView imageView = new ImageView(flowerImage);

      // Resize the ImageView, preserving its aspect ratio.
      imageView.setFitWidth(200);
      imageView.setPreserveRatio(true);

      // Put the ImageView in an HBox.
      HBox imageHBox = new HBox(imageView);

      // Center the HBox contents.
      imageHBox.setAlignment(Pos.CENTER);

      // Create the RadioButtons.
      RadioButton flowerRadio = new RadioButton("Flower");
      RadioButton sunsetRadio = new RadioButton("Sunset");

      // Select the flowerRadio control.
      flowerRadio.setSelected(true);

      // Add the RadioButtons to a ToggleGroup.
      ToggleGroup radioGroup = new ToggleGroup();
      flowerRadio.setToggleGroup(radioGroup);
      sunsetRadio.setToggleGroup(radioGroup);

      // Register an ActionEvent handler for the flowerRadio.
      flowerRadio.setOnAction(event ->
      {
         imageView.setImage(flowerImage);
      });

      // Register an event handler for the sunsetRadio.
      sunsetRadio.setOnAction(event ->
      {
         imageView.setImage(sunsetImage);
      });

      // Add the RadioButtons to a VBox.
      VBox radioVBox = new VBox(10, flowerRadio, sunsetRadio);

```

```
74         // Give the radioVBox some padding.
75         radioVBox.setPadding(new Insets(30));
76 
77         // Add everything to a VBox.
78         VBox mainVBox = new VBox(10, imageHBox, radioVBox);
79 
80         // Create a Scene with the HBox as its root node.
81         Scene scene = new Scene(mainVBox);
82 
83         // Add the Scene to the Stage.
84         primaryStage.setScene(scene);
85 
86         // Show the window.
87         primaryStage.show();
88     }
89 }
```

Let's take a closer look at the program:

- Lines 1 through 13 have the necessary JavaFX `import` statements.
- Lines 31 and 32 load the images.
- Line 35 creates an `ImageView` object, initially displaying the flower.
- In lines 38 through 45, the `ImageView` object is resized and placed in an `HBox` with center alignment.
- Lines 48 and 49 create two `RadioButton` controls named `flowerRadio` and `sunsetRadio`, and line 52 selects the `flowerRadio` control. Lines 55 through 57 add the `RadioButton` controls to a `ToggleGroup`.
- Lines 60 through 63 use a lambda expression to create and register an event handler for the `flowerRadio` control. When the event handler executes, it changes the image that is displayed by the `imageView` control to the flower image.
- Lines 66 through 69 use a lambda expression to create and register an event handler for the `sunsetRadio` control. When the event handler executes, it changes the image that is displayed by the `imageView` control to the sunset image.
- Line 72 adds the `RadioButton` controls to a `VBox`, and line 75 sets the padding for the `VBox`.
- Lines 78 through 87 perform all the necessary steps to add the controls to layout containers, create a scene, add the scene to the stage, and display the window.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.12 How do you create a mutually exclusive relationship between `RadioButton` controls?

13.13 How do you determine in code whether a `RadioButton` is selected?

13.14 In code, how do you make a `RadioButton` appear selected?

13.15 What type of event do `RadioButton` controls generate when they are clicked?

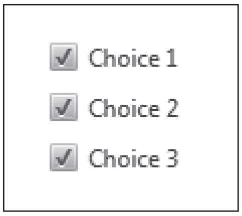
## 13.3 CheckBox Controls

**CONCEPT: CheckBox controls, which may appear alone or in groups, allow the user to make yes/no or on/off selections.**

VideoNote
CheckBox Controls

A `CheckBox` appears as a small box with a label appearing next to it. Figure 13-10 shows three `CheckBox` controls.

**Figure 13-10** CheckBox controls (Oracle Corporate Counsel)



☑ Choice 1
☑ Choice 2
☑ Choice 3

Like `RadioButtons`, `CheckBox` controls may be selected or deselected at runtime. When a `CheckBox` is selected, a small check mark appears inside the box. Although `CheckBox` controls are often displayed in groups, they are not usually grouped in a `ToggleGroup` like `RadioButtons`. This is because `CheckBox` controls are not normally used to make mutually exclusive selections. Instead, the user is allowed to select any or all of the `CheckBox` controls displayed within a group.

To create a `CheckBox` control, you will use the `CheckBox` class, which is in the `javafx.scene.control` package. Once you have written the necessary `import` statement, you will create an instance of the `CheckBox` class. The following statement shows an example of creating a `CheckBox` object that displays the text *Macaroni*:

```
CheckBox check1 = new CheckBox("Macaroni");
```

Notice we pass the string we want displayed next to the `CheckBox` as an argument to the `CheckBox` class's constructor. Optionally, you can omit the string argument to create a `CheckBox` control with no text displayed next to it. Here is an example:

```
CheckBox check1 = new CheckBox();
```

### Determining in Code Whether a CheckBox Is Selected

As with `RadioButton` controls, you use the `isSelected` method to determine whether a `CheckBox` control is selected. The method returns a `boolean` value indicating whether the `CheckBox` is selected. If the `CheckBox` is selected, the method returns `true`. Otherwise, it returns `false`. In the following code, the `check1` variable references a `CheckBox`. The `if` statement calls the `isSelected` method to determine whether the `CheckBox` is selected.

```
if (check1.isSelected())
{
    //  Code here executes if the check
    //  box is selected.
}
```

## Selecting a CheckBox in Code

You can select a `CheckBox` in code with the `CheckBox` class's `setSelected` method. The `setSelected` method takes a `boolean` argument. If you pass `true` to the method, the `CheckBox` will be selected just as if the user had clicked it. (If you pass `false` as an argument to the `setSelected` method, the `CheckBox` will be deselected.)

In the following statement, the `check1` variable references a `CheckBox`. When this statement executes, the `check1` control will be selected.

```
check1.setSelected(true);
```

Let's look at an example. Code Listing 13-16 shows the code for the PizzaToppings application, and Figure 13-11 shows how the application's window initially appears. The user checks the pizza topping items, then clicks the `Button` control to see the total cost of the selected items.

Figure 13-12 shows four different screenshots of the application running. Each screenshot shows the application after the user has made selections and clicked the `Button` control. The total of the selected items is displayed in the `Label`.

**Figure 13-11** The PizzaToppings application (Oracle Corporate Counsel)

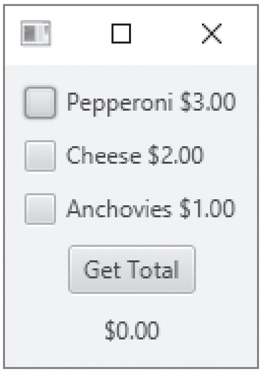
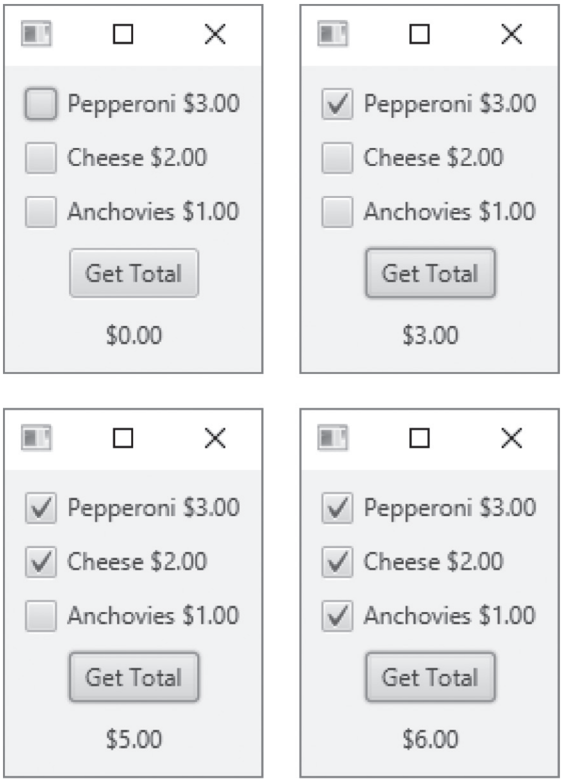


**Figure 13-12** The PizzaToppings application running (Oracle Corporate Counsel)



☐ Pepperoni $3.00
☐ Cheese $2.00
☐ Anchovies $1.00

$0.00

☑ Pepperoni $3.00
☐ Cheese $2.00
☐ Anchovies $1.00

$3.00

☑ Pepperoni $3.00
☑ Cheese $2.00
☐ Anchovies $1.00

$5.00

☑ Pepperoni $3.00
☑ Cheese $2.00
☑ Anchovies $1.00

$6.00

**Code Listing 13-16** (`PizzaToppings.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Button;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
   CheckBox Demo application
*/

public class PizzaToppings extends Application
{
```

```
// Fields
CheckBox pepperoniCheckBox;
CheckBox cheeseCheckBox;
CheckBox anchoviesCheckBox;
Label totalLabel;

public static void main(String[] args)
{
   // Launch the application.
   launch(args);
}

@Override
public void start(Stage primaryStage)
{
   // Create the CheckBoxes.
   pepperoniCheckBox = new CheckBox("Pepperoni $3.00");
   cheeseCheckBox = new CheckBox("Cheese $2.00");
   anchoviesCheckBox = new CheckBox("Anchovies $1.00");

   // Create the Button control.
   Button totalButton = new Button("Get Total");

   // Register the event handler.
   totalButton.setOnAction(new TotalButtonHandler());

   // Create a Label for the total.
   totalLabel = new Label("$0.00");

   // Put the CheckBoxes in a VBox.
   VBox checkBoxVBox = new VBox(10, pepperoniCheckBox,
                                cheeseCheckBox, anchoviesCheckBox);

   // Create another VBox to use as the root node.
   VBox mainVBox = new VBox(10, checkBoxVBox, totalButton,
                            totalLabel);

   // Set the main VBox's alignment to center.
   mainVBox.setAlignment(Pos.CENTER);

   // Set the main VBox's padding to 10 pixels.
   mainVBox.setPadding(new Insets(10));

   // Create a Scene.
   Scene scene = new Scene(mainVBox);

   // Add the Scene to the Stage.
   primaryStage.setScene(scene);
68
69     // Show the window.
70     primaryStage.show();
71  }
72
73  /**
74    Event handler class for totalButton
75  */
76
77  class TotalButtonHandler implements EventHandler<ActionEvent>
78  {
79     @Override
80     public void handle(ActionEvent event)
81     {
82        // Variable to hold the result
83        double result = 0;
84
85        // Add up the toppings.
86        if (pepperoniCheckBox.isSelected())
87        result += 3.0;
88
89        if (cheeseCheckBox.isSelected())
90        result += 2.0;
91
92        if (anchoviesCheckBox.isSelected())
93        result += 1.0;
94
95        // Display the results.
96        totalLabel.setText(String.format("$%,.2f", result));
97     }
98   }
99 }
```

Let's take a closer look at the application:

- Lines 1 through 12 have the necessary JavaFX `import` statements.
- Lines 21 through 24 declare `pepperoniCheckBox`, `cheeseCheckBox`, `anchoviesCheckBox`, and `totalLabel` as fields in the `PizzaToppings` class. The reason we are declaring them as fields (instead of declaring them as local variables in the `start` method) is because the event handler object needs to access them.
- Lines 36 through 38 create the `CheckBox` controls, and line 41 creates the `Button` control.
- Line 44 registers an instance of the `TotalButtonHandler` class as an event handler for the `Button` control.

- Line 47 creates a `Label` control to display the total.
- Lines 50 and 51 put the `CheckBox` controls in a `VBox`.
- Lines 54 and 55 create another `VBox` to serve as the root node. All of the controls are added to this `VBox`. Lines 58 and 61 set the `VBox`'s alignment to center, and the padding to 10 pixels.
- Lines 64 through 70 perform all the necessary steps to add the root node to a scene, add the scene to the stage, and display the window.
- The event handler appears in lines 77 through 98. It is an inner class named `TotalButtonHandler`. The `handle` method appears in lines 80 through 97.
  - Line 83 declares a local variable named `result`, initialized to 0, to hold the sum of the selected toppings.
  - The `if` statements in lines 86 through 93 determine which of the `CheckBox` controls is selected, and adds the selected topping amounts to the `result` variable.
  - Line 96 displays the results in the `totalLabel` control.

## Responding to `CheckBox` Clicks

Just like `Button` and `RadioButton` controls, `CheckBox` controls generate an `ActionEvent` when they are clicked. If you want an action to take place immediately when the user clicks a `CheckBox`, you have to register an event handler with the control. The process of creating and registering an event handler for a `CheckBox` is the same as the process of creating and registering an event handler for a `Button` or a `RadioButton`.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.16 How do you determine in code whether a `CheckBox` is selected?

13.17 In code, how do you make a `CheckBox` appear selected?

13.18 What type of event do `CheckBox` controls generate when they are clicked?

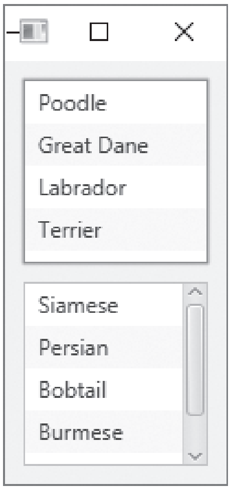
## 13.4 `ListView` Controls

**CONCEPT: The `ListView` control displays a list of items and allows the user to select an item from the list.**

VideoNote `ListView` Controls

The `ListView` control displays a list of items and allows the user to select one or more items from the list. Figure 13-13 shows an application's window with two `ListView` controls. Notice the topmost `ListView` does not have a scroll bar, but the bottom one does. A scroll bar automatically appears when a `ListView` contains more items than can be displayed in the space provided. In the figure, the top `ListView` has four items (Poodle, Great Dane, Labrador, and Terrier), and all items are displayed. The bottom `ListView` shows four items (Siamese, Persian, Bobtail, and Burmese), but because it has a scroll bar, we know there are more items in the `ListView` than those four.

**Figure 13-13** ListView examples (Oracle Corporate Counsel)



You create a `ListView` control with the `ListView` class (which is in the `javafx.scene.control` package). The following code snippet shows an example of creating a `ListView` named `dogListView`:

```
ListView<string> dogListView = new ListView<>();
```

Notice at the beginning of the statement, the notation `<string>` appears immediately after the word `ListView`. This specifies that this particular `ListView` control can hold only `String` objects. If we try to store any other type of object in this `ListView`, an error will occur when we compile the program. Also notice the diamond operator `<>` appears after `ListView` at the end of the statement. (You might recall from Chapter 7 that ArrayList objects are declared in a similar fashion.)

Once you have created a `ListView` control, you are ready to add items to it. The following statement shows how we can add strings to our `dogListView` control:

```
dogListView.getItems().addAll("Poodle", "Great Dane", "Labrador", "Terrier");
```

**NOTE:** The `ListView` control keeps its list of items in an `ObservableList` object. We discussed the `ObservableList` interface in Chapter 12. The `ListView` class's `getItems()` method returns the `ObservableList` that holds the `ListView`'s items.

You can set the preferred size of the `ListView` control with the `setPrefSize` method. You pass the preferred width and height as arguments. The following code snippet shows an example of creating a `ListView` control that is 120 pixels wide by 100 pixels high:

```
ListView<string> catListView = new ListView<>();
catListView.setPrefSize(120, 100);
catListView.getItems().addAll("Siamese", "Persian", "Bobtail",
                              "Burmese", "Tabby");
```

## Retrieving the Selected Item

You can use the `ListView` class's `getSelectionModel().getSelectedItem()` method get the item that is currently selected. For example, assume we have a `ListView` control named `listView`. The following code gets the item that is currently selected in the control, and assigns it to a `String` named `selected`:

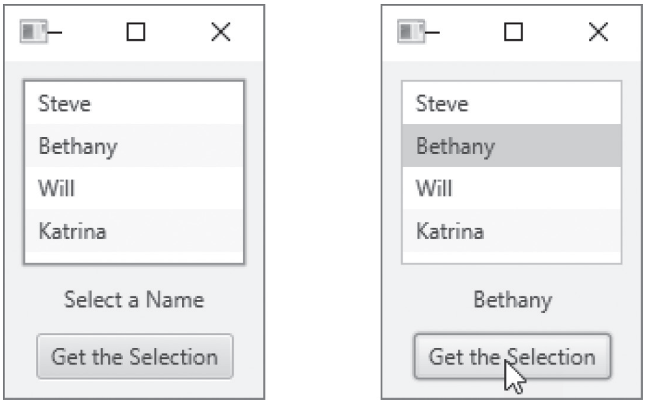
```
String selected = listView.getSelectionModel().getSelectedItem();
```

If no item is selected in the `ListView`, the method will return `null`. Code Listing 13-17 shows a complete example. It displays a list of names in a `ListView` control. When the user clicks the `Button` control, the program gets the selected name and displays it in a `Label` control. Figure 13-14 shows the program running.

**Code Listing 13-17** (`ListViewDemo1.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ListViewDemo1 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a ListView of Strings.
      ListView<string> listView = new ListView<>();
      listView.setPrefSize(120, 100);
      listView.getItems().addAll("Steve", "Bethany", "Will", "Katrina");

      // Create a Label to display the selection.
      Label selectedNameLabel = new Label("Select a Name");

      // Create a Button to get the selection.
      Button getButton = new Button("Get the Selection");

      // Create an event handler for the Button.
      getButton.setOnAction(event ->
      {
         // Get the selected name.
         String selected = listView.getSelectionModel().getSelectedItem();

         // Display the selected name in the Label.
         selectedNameLabel.setText(selected);
       });

       // Add the controls to a VBox.
       VBox vbox = new VBox(10, listView, selectedNameLabel, getButton);
       vbox.setPadding(new Insets(10));
       vbox.setAlignment(Pos.CENTER);

       // Create a Scene and display it.
       Scene scene = new Scene(vbox);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

**Figure 13-14** The ListViewDemo1.java application (Oracle Corporate Counsel)



## Retrieving the Index of the Selected Item

Internally, each item in a `ListView` control is assigned an index that marks the item's position. The first item (which is the item stored at the top of the list) has the index 0, the second item has the index 1, and so forth. You can use the `ListView` class's `getSelectionModel().getSelectedIndex()` method get the index of the item that is currently selected. For example, assume we have a `ListView` control named `listView`. The following code gets the index of the item that is currently selected in the control, and assigns it to an `int` variable named `index`:

```
int index = liststView.getSelectionModel().getSelectedIndex();
```

If no item is selected in the `ListView`, the method will return `-1`. You can use this to determine whether the user has selected an item. For example, Code Listing 13-18 is a modification of the program in Code Listing 13-17. In this version, the program displays the message "Nothing Selected" if the user clicks the `Button` control before he or she has selected an item in the `ListView` control. An example of this output is shown in Figure 13-15.

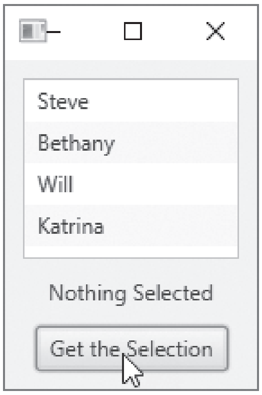
**Code Listing 13-18** `(ListViewDemo2.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ListViewDemo2 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a ListView of Strings.
      ListView<string> listView = new ListView<>();
      listView.setPrefSize(120, 100);
      listView.getItems().addAll("Steve", "Bethany", "Will", "Katrina");

      // Create a Label to display the selection.
      Label selectedNameLabel = new Label("Select a Name");

      // Create a Button to get the selection.
      Button getButton = new Button("Get the Selection");

      // Create an event handler for the Button.
      getButton.setOnAction(event ->
      {
         // Determine whether an item is selected.
         if (listView.getSelectionModel().getSelectedIndex() != -1)
         {
```

```
39             // Get the selected name.
40             String selected =
41                     listView.getSelectionModel().getSelectedItem();
42 
43             // Display the selected name in the Label.
44             selectedNameLabel.setText(selected);
45          }
46          else
47          {
48             selectedNameLabel.setText("Nothing Selected");
49          }
50       });
51 
52       // Add the controls to a VBox.
53       VBox vbox = new VBox(10, listView, selectedNameLabel, getButton);
54       vbox.setPadding(new Insets(10));
55       vbox.setAlignment(Pos.CENTER);
56 
57       // Create a Scene and display it.
58       Scene scene = new Scene(vbox);
59       primaryStage.setScene(scene);
60       primaryStage.show();
61    }
62 }
```

Let's take a closer look at the program. In the `Button` control's event handler, the `if` statement that begins in line 37 gets the index of the selected item. If that value is *not* −1, then an item has been selected, and the selected item is displayed (in line 44). However, if the selected item index is −1, then nothing is selected in the `ListView`, and the program jumps to the `else` clause in line 46.

**Figure 13-15** The ListViewDemo2.java application (Oracle Corporate Counsel)



## Responding to Item Selection with an Event Handler

When the user selects an item in a `ListView`, a *change event* occurs. If you want the program to immediately perform an action when the user selects an item in a `ListView`, you can write an event handler that responds to the change event. Assuming `listView` is the name of a `ListView` control, the following code snippet shows how to use a lambda expression to register such an event handler:

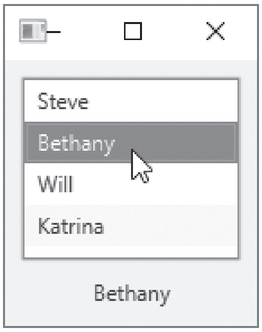
```
listView.getSelectionModel().selectedItemProperty().addListener(event ->
{
    //  Write event handling code here ...
});
```

For example, Code Listing 13-19 is another modification of the program in Code Listing 13-17. This version of the application does not have a `Button` control. Instead, the selected item is displayed immediately after the user makes his or her selection. An example of the program's output is shown in Figure 13-16.

**Code Listing 13-19** (`ListViewDemo3.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ListViewDemo3 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a ListView of Strings.
      ListView<string> listView = new ListView<>();
      listView.setPrefSize(120, 100);
      listView.getItems().addAll("Steve", "Bethany", "Will", "Katrina");

      // Create a Label to display the selection.
      Label selectedNameLabel = new Label("Select a Name");

      // Create an event handler for the ListView control.
      listView.getSelectionModel().selectedItemProperty().addListener(event ->
      {
         // Get the selected name.
         String selected = listView.getSelectionModel().getSelectedItem();

         // Display the selected name in the Label.
         selectedNameLabel.setText(selected);
      });

      // Add the controls to a VBox.
      VBox vbox = new VBox(10, listView, selectedNameLabel);
      vbox.setPadding(new Insets(10));
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-16** The ListViewDemo3.java application (Oracle Corporate Counsel)



## Adding Items versus Setting Items

The `ListView` control's `getItems().addAll()` method adds items to the `ListView`. If the `ListView` already contains items, the `getItems().addAll()` method will not erase the existing items, but will add the new items to the existing list. As a result, the `ListView` will contain the old items plus the new items.

Sometimes this is the behavior you want, but in other situations, you might want to replace the existing contents of a `ListView` with an entirely new list of items. When that is the case, you can use the `getItems().setAll()` method. The `getItems().setAll()` method replaces a `ListView` control's existing items with a new list of items.

The following code snippet demonstrates how to create a `ListView` containing one list of items, then replace those items with another list:

```
// Create a ListView.
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday");

// Replace the existing items with a new list.
listView.getItems().setAll("Thursday", "Friday", "Saturday");
```

After this code executes, the `listView` control will contain the items “Thursday”, “Friday”, and “Saturday”.

## Initializing a `ListView` with an Array or an `ArrayList`

In some situations, you might have an array or an `ArrayList` containing items you want to display in a `ListView` control. When this is the case, you can convert the array or `ArrayList` to an `ObservableList`, then pass the `ObservableList` as an argument to the `ListView` class’s constructor.

The `FXCollections` class (which is in the `javafx.collections` package) has a static method named `observableArrayList` that takes an array or an `ArrayList` as an argument, and returns an `ObservableList` containing the same items. The following code shows an example of creating a `ListView` control that displays the items in a `String` array:

```
// Create a String array.
String[] strArray = {"Monday", "Tuesday", "Wednesday"};

// Convert the String array to an ObservableList.
ObservableList<String> strList =
       FXCollections.observableArrayList(strArray);

// Create a ListView control.
ListView<String> listView = new ListView<>(strList);
```

The following code snippet shows the same example, but this time converting an `ArrayList` to an `ObservableList`:

```
// Create an ArrayList of Strings.
ArrayList<String> strArrayList = new ArrayList<>();
strArrayList.add("Monday");
strArrayList.add("Tuesday");
strArrayList.add("Wednesday");

// Convert the ArrayList to an ObservableList.
ObservableList<String> strList =
       FXCollections.observableArrayList(strArrayList);

// Create a ListView control.
ListView<String> listView = new ListView<>(strList);
```

You can also populate a `ListView` by passing an `ObservableList` to the `ListView` class’s `getItems().addAll()` and `getItems().setAll()` methods. Here is an example that uses the `getItems().addAll()` method to populate a `ListView` with the items in a `String` array:

```
// Create a String array.
String[] strArray = {"Monday", "Tuesday", "Wednesday"};

// Convert the String array to an ObservableList.
ObservableList<String> strList =
       FXCollections.observableArrayList(strArray);

// Create a ListView control.
ListView<String> listView = new ListView<>();

// Populate the ListView control:
listView.getItems().addAll(strList);
```

Here is an example that uses the `getItems().setAll()` method to populate a `ListView` with the items in an ArrayList of `Strings`:

```
// Create an ArrayList of Strings.
ArrayList<String> strArrayList = new ArrayList<>();
strArrayList.add("Monday");
strArrayList.add("Tuesday");
strArrayList.add("Wednesday");

// Convert the ArrayList to an ObservableList.
ObservableList<String> strList =
       FXCollections.observableArrayList(strArrayList);

// Create a ListView control.
ListView<String> listView = new ListView<>();

// Populate the ListView control:
listView.getItems().setAll(strList);
```

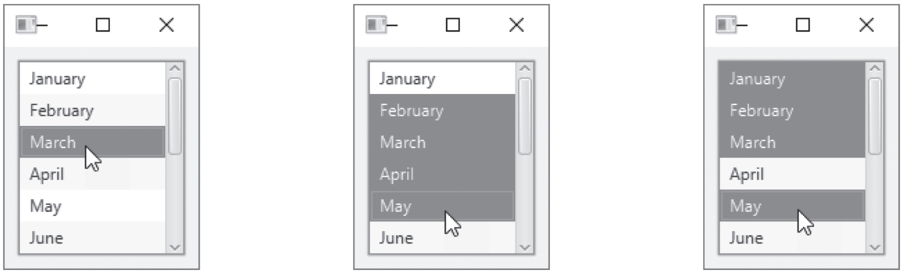
## Selection Modes

The `ListView` control can operate in either of the following selection modes:

- **Single Selection Mode.** In this mode, only one item can be selected at a time. When an item is selected, any other item that is currently selected is deselected. This is the default selection mode
- **Multiple Interval Selection Mode.** In this mode, multiple items may be selected. To select multiple items that are consecutive, the user clicks the first item, holds down the Shift key on the keyboard, then clicks the last item. To select items that are not consecutive, the user holds down the Ctrl key on the keyboard (or the Command key on a Mac keyboard), then clicks each item.

Figure 13-17 shows examples of a `ListView` control in single selection mode and multiple selection mode.

**Figure 13-17** Selection modes (Oracle Corporate Counsel)



In single selection mode, only one item can be selected at a time.

In multiple selection mode, multiple consecutive items can be selected.

In multiple selection mode, multiple nonconsecutive items can be selected.

By default, the `ListView` control is in single selection mode. You change the selection mode with the control's `getSelectionModel().setSelectionMode()` method. The method accepts one of the following enum constants:

- `SelectionMode.MULTIPLE`
- `SelectionMode.SINGLE`

Note the `SelectioMode` enum is in the `javafx.scene.control` package. Assuming that `listView` is the name of a `ListView` control, the following code shows how to change the control to multiple selection mode:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

## Retrieving Multiple Selected Items

When a `ListView` control is in multiple selection mode, the user can select more than one item. The `getSelectionModel().getSelectedItem()` method will return only the last item selected. Likewise, the `getSelectionModel().getSelectedIndex()` method will return only the index of the last item selected. To get all of the selected items and their indices, you need to use the following methods:

- `getSelectionModel().getSelectedItems()`—This method returns a read-only `ObservableList` of the selected items
- `getSelectionModel().getSelectedIndices()`—This method returns a read-only `ObservableList` of the integer indices of the selected items

Code Listing 13-20 demonstrates how to retrieve multiple selected items. The program displays two `ListView` controls, named `listView1` and `listView2`. The user selects items from `listView1` and then, when the user clicks the `Button` control, the selected items are displayed in `listView2`. Figure 13-18 shows an example of the application as it is running. The image on the left shows the application's window as it initially appears, and the image on the right shows the window after the user has selected several items from `listView1` and clicked the `Button` control.

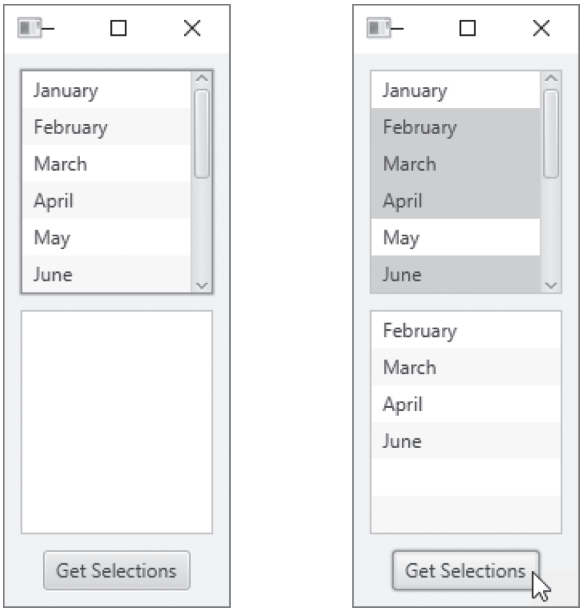
**Code Listing 13-20** (`ListViewDemo4.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.collections.ObservableList;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ListViewDemo4 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the ListView sizes
      final double WIDTH = 120, HEIGHT = 140;

      // Create a ListView of the names of the months.
      ListView<String> listView1 = new ListView<>();
      listView1.setPrefSize(WIDTH, HEIGHT);
      listView1.getSelectionModel().setSelectionMode(
          SelectionMode.MULTIPLE);
      listView1.getItems().addAll(
           "January", "February", "March", "April", "May",
           "June", "July", "August", "September", "October",
           "November", "December");

      // Create an empty ListView to show the selections.
      ListView<String> listView2 = new ListView<>();
      listView2.setPrefSize(WIDTH, HEIGHT);

41       // Create a Button to get the selections.
42       Button getButton = new Button("Get Selections");
43 
44       // Register an event handler for the Button.
45       getButton.setOnAction(event ->
46       {
47          // Get the ObservableList of selected items.
48          ObservableList<String> selections =
49             listView1.getSelectionModel().getSelectedItems();
50 
51          // Add the selections to the 2nd ListView.
52          listView2.getItems().setAll(selections);
53       });
54 
55       // Add the controls to a VBox.
56       VBox vbox = new VBox(10, listView1,
57                            listView2, getButton);
58       vbox.setPadding(new Insets(10));
59       vbox.setAlignment(Pos.CENTER);
60 
61       // Create a Scene and display it.
62       Scene scene = new Scene(vbox);
63       primaryStage.setScene(scene);
64       primaryStage.show();
65    }
66 }
```

Let's take a closer look at the program:

- Line 25 declares constants for the `ListView` dimensions.
- Lines 28 through 35 create the `listView1` control, set its size, set its selection mode to `SelectionMode.MULTIPLE`, and populate it with the names of the months.
- Lines 38 and 39 create the `listView2` control and set its size.
- Line 42 creates a `Button` control, and lines 45 through 53 register an event handler for the `Button`.
    - ❍ Lines 48 and 49 get the `ObservableList` of items that are selected in the `listView1` control.
    - ❍ Line 52 displays the `ObservableList` in the `listView2` control.
- Lines 56 through 59 create a `VBox` and add all of the controls to it.
- Lines 62 through 64 create a scene, set the scene to the stage, and display the application's window.

**Figure 13-18** The ListViewDemo4.java application (Oracle Corporate Counsel)



The window as it initially appears

The window after the user has selected four items and clicked the button

## Working With the Elements of an `ObservableList`

An `ObservableList` has much in common with the `ArrayList` class that you learned about in Chapter 7. If you need to work with the individual elements in an `ObservableList`, everything that you have already learned about `ArrayLists` also applies to `ObservableLists`. For example:

- You can use the enhanced `for` loop to iterate over an `ObserveableList`.
- You can use the `get` method to get a specific element in an `ObservableList`.
- You can use the `size` method to get the number of elements in an `ObservableList`.
- You can use the `add` method to add an element to an `ObservableList` (as long as the `ObservableList` is not read-only).
- You can use the `remove` method to remove an element from an `ObservableList`.
- You can use the `set` method to replace an element in an `ObservableList`.

The following code snippet shows a different version of the `getButton` control's event handler in Code Listing 13-20. In this version of the code, we are using an enhanced `for` loop in line 52 to iterate over the `ObservableList`, adding one item at a time to the `listView2` control (line 53):

```
44      // Register an event handler for the Button.
45      getButton.setOnAction(event ->
46      {
47         // Get the ObservableList of selected items.
48         ObservableList<String> selections =
49            listView1.getSelectionModel().getSelectedItems();
50
51        // Add the selections to the 2nd ListView.
52        for (String str : selections)
53            listView2.getItems().addAll(str);
54      });
```

The following code snippet shows yet another version of the `getButton` control's event handler in Code Listing 13-20. In this version of the code, we are using a regular `for` loop in line 52 to iterate over the `ObservableList`, adding one item at a time to the `listView2` control (line 53):

```
44     // Register an event handler for the Button.
45     getButton.setOnAction(event ->
46     {
47        // Get the ObservableList of selected items.
48        ObservableList<String> selections =
49           listView1.getSelectionModel().getSelectedItems();
50
51        // Add the selections to the 2nd ListView.
52        for (int i = 0; i < selections.size(); i++)
53           listView2.getItems().addAll(selections.get(i));
54     });
```

Obviously, neither of these approaches are better than the original approach shown in Code Listing 13-20, but these examples illustrate how you can iterate over an `ObservableList` and work with its individual elements.

## Converting an `ObservableList` to an Array

Sometimes, you might need to get the `ObservableList` of selected items from a `ListView` control, then convert that `ObservableList` to an array. `ObservableList` objects have a method named `toArray` that returns an array containing the items in the list. Before the `toArray` method can convert the `ObservableList` to an array, however, it must know the data type of the array. You specify the data type by passing an empty array of the correct data type as an argument to the `toArray` method. Let's look at an example. In the following code snippet, assume `listView` is a `ListView` control containing strings:

```
// Get the selected items as an ObservableList.
ObservableList<String> selections =
       listView.getSelectionModel().getSelectedItems();

// Convert the ObservableList to an array of Strings.
String[] itemArray = selections.toArray(new String[0]);
```

In this code, the first statement gets an `ObservableList` containing the selected items in the `listView` control. The second statement converts the `ObservableList` to a `String` array, and assigns the array to the `itemArray` variable. The empty `String` array we are passing as an argument to the `toArray` method tells the method to return a `String` array.

## Using Code to Select an Item in a `ListView`

When you create a `ListView` control, none of the items in the `ListView` are initially selected by default. In code, you can use any of the methods listed in Table 13-5 to select an item or items in a `ListView` control.

**Table 13-5** `ListView` methods to select items

| Method | Description |
|---|---|
| `getSelectionModel().selectFirst()` | Selects the first item in the `ListView`. |
| `getSelectionModel().selectLast()` | Selects the last item in the `ListView`. |
| `getSelectionModel().selectAll()` | Selects all of the items in the `ListView`. |
| `getSelectionModel().selectIndices(`*`index...`*`)` | Pass one or more `int` indices as arguments, and this method selects each corresponding item in the `ListView`. |
| `getSelectionModel().selectRange(`*`start`*`, `*`end`*`)` | This method selects a range of items in the `ListView`, beginning at the *`start`* index, and ending at the *`end`* index. Both arguments are `int`s. |

For example, the following code creates a `ListView` and selects the first item:

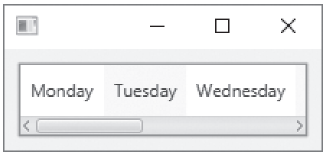
```
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday");
listView.getSelectionModel().selectFirst();
```

## `ListView` Orientation

By default, `ListView` controls are vertically oriented. You can set the orientation to either vertical or horizontal with the `ListView` class's `setOrientation` method. When you call the method, you pass it one of the following enum constants: `Orientation.VERTICAL` or `Orientation.HORIZONTAL`. (The `Orientation` enum is in the `javafx.geometry` package.) The following code snippet shows how to create a `ListView` control and set its orientation to horizontal. Figure 13-19 shows an example of how the `ListView` will appear.

```
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday",
                           "Thursday", "Friday", "Saturday",
                           "Sunday");
listView.setOrientation(Orientation.HORIZONTAL);
listView.setPrefSize(200, 50);
```

**Figure 13-19** A horizontal `ListView` (Oracle Corporate Counsel)



## Creating ListViews of Objects Other Than `String`

All of the examples we have discussed so far use `ListView`s of `String`s. You can store other types of objects in a `ListView` as well. The following code snippet creates a `ListView` that displays the integers 1 through 5:

```
ListView<Integer> listView = new ListView<>();
listView.getItems().addAll(1, 2, 3, 4, 5);
```

Notice in the first statement, the data type of the `ListView` is `ListView<Integer>`. If you want to display values of one of the primitive types, such as `int`, `double`, `float`, and so on, you must specify the type's corresponding wrapper class in the `ListView` declaration. For example, the following code snippet creates a `ListView` that displays values of the `double` type:

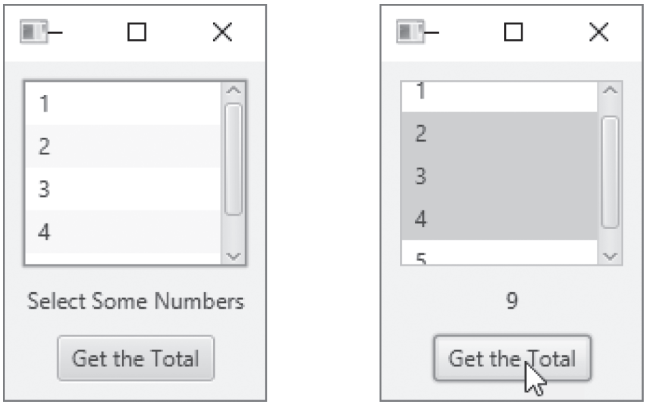
```
ListView<Double> listView = new ListView<>();
listView.getItems().addAll(1.0, 2.0, 3.0, 4.0, 5.0);
```

Code Listing 13-21 shows a complete example. As shown in Figure 13-20, this program creates a `ListView` of the integer values 1 through 5. The `ListView` is in multiple selection mode, so the user can select multiple values. When the user clicks the `Button` control, the program calculates the sum of the selected integers, and displays the sum in the `Label` control.

**Code Listing 13-21** (`ListViewDemo5.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.collections.ObservableList;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ListViewDemo5 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the ListView width and height
      final double WIDTH = 120.0, HEIGHT = 100.0;

      // Create a ListView of Integers.
      ListView<Integer> listView = new ListView<>();
      listView.setPrefSize(WIDTH, HEIGHT);
      listView.getItems().addAll(1, 2, 3, 4, 5);
      listView.getSelectionModel().setSelectionMode(
          SelectionMode.MULTIPLE);

      // Create a Label to display the selection.
      Label outputLabel = new Label("Select Some Numbers");

      // Create a Button to get the selection.
      Button totalButton = new Button("Get the Total");

      // Create an event handler for the Button.
      totalButton.setOnAction(event ->
      {
         // Determine whether an item is selected.
         if (listView.getSelectionModel().getSelectedIndex() != -1)
         {
            // Get the ObservableList of selected items.
            ObservableList<Integer> selections =
            listView.getSelectionModel().getSelectedItems();

            // Accumulator variable
            int total = 0;

            // Get the sum of the numbers.
            for (int num : selections)
            total += num;

            // Display the total in the Label.
            outputLabel.setText(Integer.toString(total));
         }
         else
         {
            outputLabel.setText("Nothing Selected");
         }
      });

      // Add the controls to a VBox.
      VBox vbox = new VBox(10, listView, outputLabel, totalButton);
      vbox.setPadding(new Insets(10));
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-20** The ListViewDemo5.java application (Oracle Corporate Counsel)



The window as it initially appears

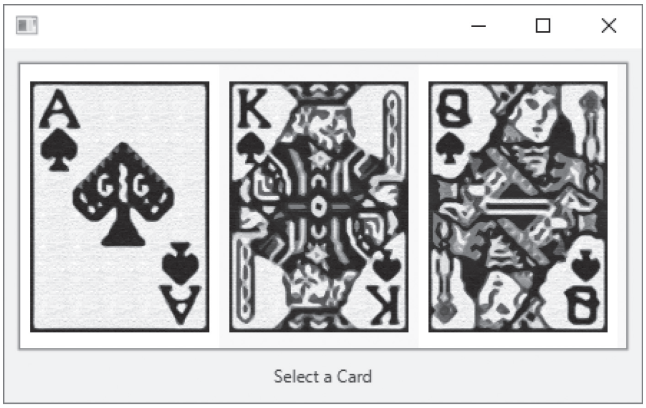
The window after the user has selected four items and clicked the button

Code Listing 13-22 demonstrates how you can store `ImageView` controls in a `ListView`. As shown in Figure 13-21, this program displays the images of three poker cards in a horizontal `ListView`. When the user clicks one of the card images, the program displays the name of the card in a `Label` control.

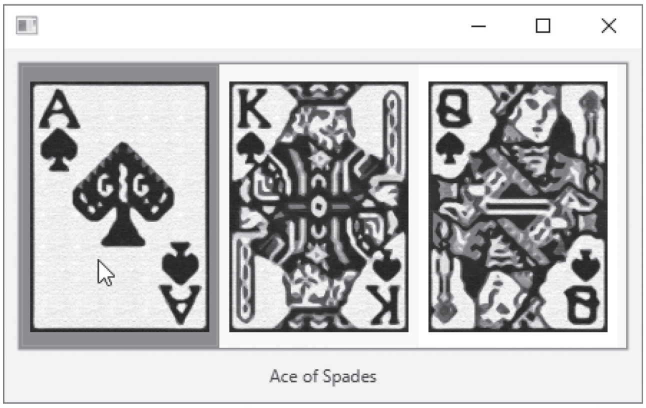
**Code Listing 13-22** (`ListViewDemo6.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;

public class Test extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the ListView dimensions
      final double WIDTH = 425.0, HEIGHT = 200.0;

      // Create the Image objects.
      Image aceSpadesImage = new Image("file:Ace_Spades.png");
      Image kingSpadesImage = new Image("file:King_Spades.png");
      Image queenSpadesImage = new Image("file:Queen_Spades.png");

      // Create the ImageView controls.
      ImageView aceSpadesIV = new ImageView(aceSpadesImage);
      ImageView kingSpadesIV = new ImageView(kingSpadesImage);
      ImageView queenSpadesIV = new ImageView(queenSpadesImage);

      // Create a ListView for the ImageView objects.
      ListView<ImageView> listView = new ListView<>();
      listView.setPrefSize(WIDTH, HEIGHT);
      listView.setOrientation(Orientation.HORIZONTAL);
      listView.getItems().addAll(aceSpadesIV, kingSpadesIV,
                                 queenSpadesIV);

      // Array with the names of the cards
      String[] cardNames = {"Ace of Spades", "King of Spades",
                            "Queen of Spades"};

      // Create a Label to display the selection.
      Label outputLabel = new Label("Select a Card");

      // Create an event handler for the ListView control.
      listView.getSelectionModel().selectedItemProperty().addListener(event ->
      {
         // Get the index of the selected card.
         int index = listView.getSelectionModel().getSelectedIndex();

         // Display the selected card name in the Label.
         outputLabel.setText(cardNames[index]);
      });

      // Add the controls to a VBox.
      VBox vbox = new VBox(10, listView, outputLabel);
      vbox.setPadding(new Insets(10));
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-21** The ListViewDemo6.java application (Oracle Corporate Counsel)



The window as it initially appears



The window after the user has clicked a card image

Let's take a closer look at the program:

- Line 25 declares constants for the size of the `ListView` control.
- Lines 28 through 30 load the card images and create `Image` objects.
- Lines 33 through 35 create `ImageView` controls for the card images.
- Line 38 creates a `ListView` control to display the card images. Notice the `ListView` data type is `ListView<ImageView>`.
- Line 39 sets the size of the `ListView`, and line 40 sets the `ListView`'s orientation to horizontal.
- Lines 41 and 42 add the `ImageView` controls to the `ListView`.
- Lines 45 and 46 declare a `String` array containing the names of the cards. Notice the following parallel relationship exists between the names of the cards in the array, and the `ImageView` controls in the `ListView`:
    - The string "Ace of Spades" is at subscript 0 in the array, and the image of the Ace of Spades is at index 0 in the `ListView`.
    - The string "King of Spades" is at subscript 1 in the array, and the image of the King of Spades is at index 1 in the `ListView`.
    - The string "Queen of Spades" is at subscript 2 in the array, and the image of the Queen of Spades is at index 2 in the `ListView`.

- Line 49 creates a `Label` control to display the name of the selected card.
- Lines 52 through 59 register an event handler with the `ListView` control. This event handler will execute any time the user clicks an item in the `ListView`:
    - Line 55 gets the index of the card image that was selected in the `ListView`.
    - Line 58 displays the corresponding string from the cardNames array. This is the name of the card that was selected in the `ListView`.
- Lines 62 through 64 put all of the controls in a `VBox`, and set the `VBox`'s alignment and padding.
- Lines 67 through 69 create a scene, put the scene on the stage, and display the application's window.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

13.19 How do you set the size of a `ListView`?

13.20 How do you retrieve the selected item from a `ListView` control?

13.21 What is single selection mode?

13.22 What is multiple interval selection mode?

13.23 How do you set the orientation of a `ListView` control?

## 13.5 ComboBox Controls

**CONCEPT: A `ComboBox` allows the user to select an item from a drop-down list.**

VideoNote ComboBox Controls

A `ComboBox` presents a list of items from which the user may select. Unlike a `ListView`, a `ComboBox` presents its items in a drop-down list. You create a `ComboBox` control with the `ComboBox` class (which is in the `javafx.scene.control` package). The following code snippet shows an example of creating a `ComboBox` named `nameComboBox`:

```
ComboBox<String> nameComboBox = new ComboBox<>();
```

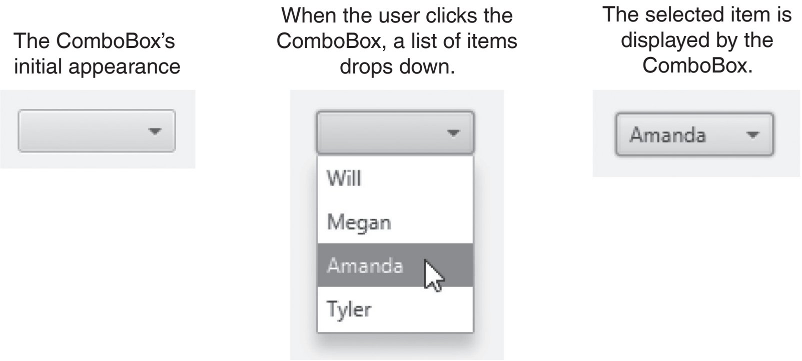
Notice at the beginning of the statement, the notation `<String>` appears immediately after the word `ComboBox`. This specifies that this particular `ComboBox` control can hold only `String` objects. If we try to store any other type of object in this `ComboBox`, an error will occur when we compile the program. Also notice the diamond operator `<>` appears after `ComboBox` at the end of the statement. (You might recall from Chapter 7 that ArrayList objects are declared in a similar fashion.)

Once you have created a `ComboBox` control, you are ready to add items to it. The following statement shows how we can add strings to our `nameComboBox` control:

```
nameComboBox.getItems().addAll("Will", "Megan", "Amanda", "Tyler");
```

The `nameComboBox` control created by this code will initially appear as the button shown on the left in Figure 13-22. The button displays the item that is currently selected. When the user clicks the button, the drop-down list appears as shown in the image in the middle of the figure. When the user selects an item from the list, it will appear on the `ComboBox`'s button, as shown in the image on the right in the figure.

**Figure 13-22** The ComboBox control (Oracle Corporate Counsel)


The ComboBox's initial appearance
When the user clicks the ComboBox, a list of items drops down.
Will
Megan
Amanda
Tyler
The selected item is displayed by the ComboBox.
Amanda

**NOTE:** Just like the `ListView` control, the `ComboBox` control keeps its list of items in an `ObservableList` object. We discussed the `ObservableList` interface in Chapter 12. The `ComboBox` class's `getItems()` method returns the `ObservableList` that holds the `ComboBox`'s items.

Table 13-6 lists some of the most often used `ComboBox` methods.

**Table 13-6** Some of the ComboBox methods

| Method | Description |
|---|---|
| `getValue()` | Returns the item that is currently selected in the `ComboBox`. |
| `setValue(`*`value`*`)` | Selects *`value`* in the `ComboBox`. |
| `setVisibleRowCount(`*`count`*`)` | The *`count`* argument is an `int`. Sets the number of rows, or items, to display in the drop-down list. |
| `setEditable(`*`value`*`)` | The *`value`* argument is a `boolean`. If *`value`* is `true`, the `ComboBox` will be editable. If *`value`* is `false`, the `ComboBox` will be uneditable. |
| `show()` | Displays the drop-down list of items. |
| `hide()` | Hides the drop-down list of items. |
| `isShowing()` | Returns `true` or `false` to indicate whether the drop-down list is visible. |

## Retrieving the Selected Item

You can use the `ComboBox` class's `getValue()` method get the item that is currently selected. For example, assume we have a `ComboBox` control named `comboBox`. The following code gets the item that is currently selected in the control, and assigns it to a `String` named `selected`:

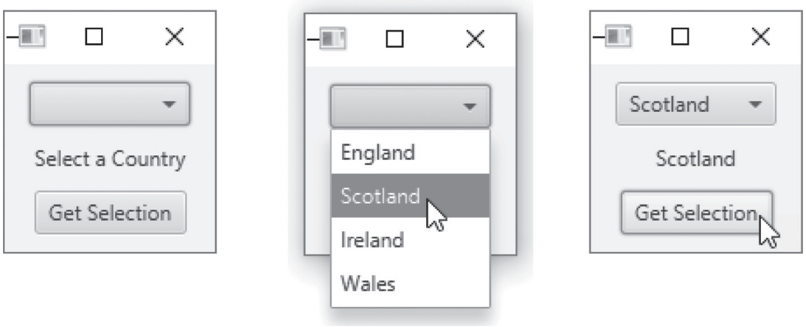
```
String selected = comboBox.getValue();
```

If no item is selected in the `ComboBox`, the method will return `null`. Code Listing 13-23 shows a complete example. It displays a list of countries in a `ComboBox` control. The user selects a country, then clicks the `Button` control. The `Button`'s event handler gets the selected country and displays it in a `Label` control. Figure 13-23 shows the program running.

**Code Listing 13-23** (`ComboBoxDemo1.java`)

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ComboBoxDemo1 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a ComboBox.
      ComboBox<string> comboBox = new ComboBox<>();
      comboBox.getItems().addAll("England", "Scotland",
                                 "Ireland", "Wales");

      // Create a Label.
      Label outputLabel = new Label("Select a Country");

      // Create a Button.
      Button button = new Button("Get Selection");
      button.setOnAction(event ->
      {
         outputLabel.setText(comboBox.getValue());
      });

      // Add the controls to a VBox.
      VBox vbox = new VBox(10, comboBox, outputLabel, button);
      vbox.setPadding(new Insets(10));
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-23** The ComboBoxDemo1.java application (Oracle Corporate Counsel)



## Responding to ComboBox Item Selection with an Event Handler

When the user selects an item in a `ComboBox`, an `ActionEvent` occurs. If you want the program to immediately perform an action when the user selects an item in a `ListView`, you can write an event handler that responds to the action event. Assuming `comboBox` is the name of a `ComboBox` control, the following code snippet shows how to use a lambda expression to register such an event handler:

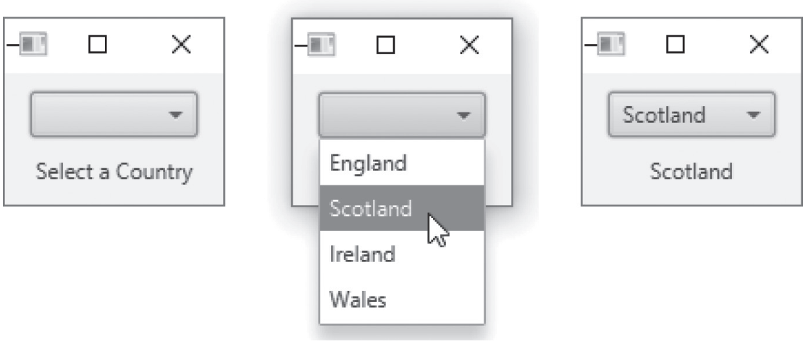
```
comboBox.setOnAction(event ->
{
   //  Write event handling code here...
});
```

For example, Code Listing 13-24 is a modification of the program in Code Listing 13-23. This version of the application does not have a `Button` control. Instead, the selected item is displayed immediately after the user makes his or her selection. An example of the program's output is shown in Figure 13-24.

**Code Listing 13-24** (`ComboBoxDemo2.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class ComboBoxDemo2 extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a ComboBox.
      ComboBox<String> comboBox = new ComboBox<>();
      comboBox.getItems().addAll("England", "Scotland",
                                 "Ireland", "Wales");

      // Create a Label.
      Label outputLabel = new Label("Select a Country");

      // Register an event handler for the ComboBox.
      comboBox.setOnAction(event ->
      {
         outputLabel.setText(comboBox.getValue());
      });

      // Add the controls to a VBox.
      VBox vbox = new VBox(10, comboBox, outputLabel);
      vbox.setPadding(new Insets(10));
      vbox.setAlignment(Pos.CENTER);

      // Create a Scene and display it.
      Scene scene = new Scene(vbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-24** The ComboBoxDemo2.java application (Oracle Corporate Counsel)



## Editable ComboBoxes

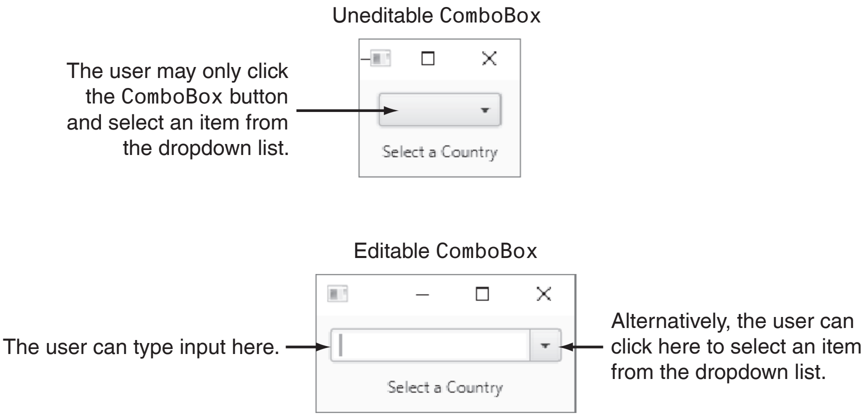
An editable `ComboBox` allows the user to select an item from a drop-down list, or type input into a field that is similar to a `TextField`. By default, `ComboBox` controls are uneditable, which means the user cannot type input into the control; he or she can only select items from a drop-down list.

The `ComboBox` class has a method named `setEditable` that takes a `boolean` value as its argument. If you pass `true`, the `ComboBox` becomes editable. If you pass `false`, the `ComboBox` becomes uneditable. Here is an example code snippet that demonstrates this:

```
ComboBox<String> comboBox = new ComboBox<>();
comboBox.setEditable(true);
comboBox.getItems().addAll("England", "Scotland",
                           "Ireland", "Wales");
```

Figure 13-25 compares an uneditable `ComboBox` with an editable `ComboBox`.

**Figure 13-25** Uneditable versus Editable ComboBoxes (Oracle Corporate Counsel)



The user may only click the ComboBox button and select an item from the dropdown list.

The user can type input here.

Alternatively, the user can click here to select an item from the dropdown list.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.24 In what type of object does the `ComboBox` control keep its list of items?

13.25 How do you retrieve the selected item from a `ComboBox`?

13.26 What type of event do `ComboBox` controls generate when they are clicked?

13.27 What is the difference between an editable `ComboBox` and an uneditable `ComboBox`?

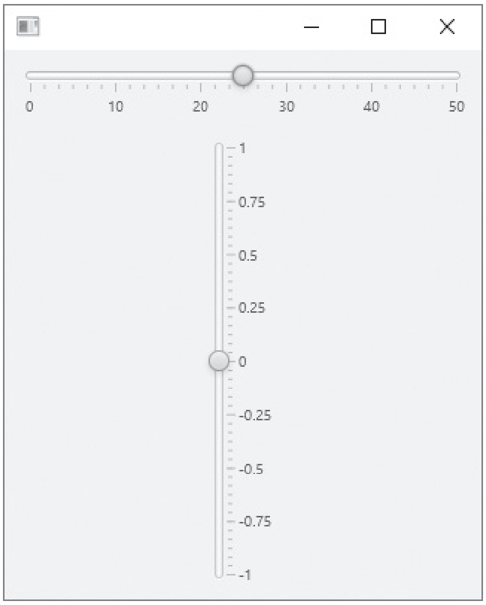
## 13.6 `Slider` Controls

**CONCEPT: A `Slider` is a control that allows the user to graphically adjust a number within a range of values.**

VideoNote
`Slider` Controls

`Slider` controls display an image of a “slider knob” that can be dragged along a track. Sliders can be horizontally or vertically oriented, as shown in Figure 13-26. You create `Slider` controls with the `Slider` class, which is in the `javafx.scene.control` package.

**Figure 13-26** A horizontal and a vertical `Slider` (Oracle Corporate Counsel)



A `Slider` is designed to represent a range of numeric values. At one end of the `Slider` is the range’s minimum value, and at the other end is the range’s maximum value. In Figure 13-26, the horizontal `Slider`’s minimum value is 0, and its maximum value is 50. The vertical `Slider`’s minimum value is −1, and its maximum value is 1.

`Slider` controls have a numeric value, and as the user moves the knob along the track, the numeric value is adjusted accordingly. Notice the `Slider` controls in Figure 13-26 have accompanying tick marks. Starting at 0, and then at every 10th value, a major tick mark is displayed along with a label indicating the value at that tick mark. Between the major tick marks are minor tick marks. In this example, there are five minor tick marks displayed between the major tick marks.

The `Slider` class has two constructors. The no-arg constructor creates a `Slider` with a minimum value of 0, and a maximum value of 100. The `Slider`’s initial value will be set to 0, and no tick marks will be displayed. Here is a statement that creates a `Slider` using the no-arg constructor:

```
Slider slider = new Slider();
```

The second constructor accepts three `double` arguments: the minimum value, the maximum value, and the initial value. The following statement creates a `Slider` with the minimum value 0, the maximum value 50, and the initial value 25:

```
Slider slider = new Slider(0.0, 50.0, 25.0);
```

The appearance of tick marks, their spacing, and the appearance of labels can be controlled through methods in the `Slider` class. Table 13-7 lists some of the methods you will use most often.

**Table 13-7** Some of the `Slider` class methods

| Method | Description |
|---|---|
| `setOrientation(`*`orientation`*`)` | Sets the `Slider`'s orientation. The argument should be one of the following enum constants: `Orientation.HORIZONTAL` or `Orientation.VERTICAL`. (The `Orientation` enum is in the `javafx.geometry` package. By default, `Slider` controls are horizontal.) |
| `setShowTickMarks(`*`value`*`)` | The *`value`* argument is a `boolean` value. If you pass `true`, the `Slider` will display tick marks. If you pass `false`, no tick marks will be displayed. (By default, no tick marks are displayed.) |
| `setMajorTickUnit(`*`value`*`)` | The *`value`* argument is a `double`. Sets the major tick mark spacing. (The default value is 25.) |
| `setMinorTickCount(`*`value`*`)` | The *`value`* argument is an `int`. Sets the number of minor tick marks to place between the major tick marks. (The default value is 3.) |
| `setMax(`*`value`*`)` | The *`value`* argument is a `double`. Sets the `Slider`'s maximum value. (By default, the maximum value is 100.) |
| `setMin(`*`value`*`)` | The *`value`* argument is a `double`. Sets the `Slider`'s minimum value. (By default, the minimum value is 0.) |
| `setSnapToTicks(`*`value`*`)` | The *`value`* argument is a `boolean` value. If you pass `true`, the `Slider` knob will always snap to the nearest tick mark when it is moved. If you pass `false`, the `Slider` knob will not snap to tick marks. (By default, the `Slider` knob does not snap.) |
| `setShowTickLabels(`*`value`*`)` | The *`value`* argument is a `boolean` value. If you pass `true`, numeric labels are displayed at the major tick marks. If you pass `false`, no labels will be displayed. (By default, no labels are displayed.) |
| `setBlockIncrement(`*`value`*`)` | The *`value`* argument is a `double`. Sets the amount by which the `Slider`'s knob is moved when the user clicks the `Slider`'s track, or uses the keyboard to move the knob. (The default value is 10.) |
| `setValue(`*`value`*`)` | The *`value`* argument is a `double`. Sets the `Slider`'s current value. |
| `getValue()` | Returns the `Slider`'s current value, as a `double`. |

The following code snippet creates the horizontal `Slider` that is shown in Figure 13-26:

```
1   Slider hSlider = new Slider(0.0, 50.0, 25.0);
2   hSlider.setShowTickMarks(true);
3   hSlider.setMajorTickUnit(10);
4   hSlider.setMinorTickCount(5);
5   hSlider.setShowTickLabels(true);
6   hSlider.setPrefWidth(300.0);
```

Let's take a closer look at the code:

- Line 1 creates a `Slider` with a minimum value of 0, and a maximum value of 50. The `Slider`'s initial value is 25. By default, the `Slider`'s orientation is horizontal.
- Line 2 causes the `Slider` to display tick marks.
- Line 3 sets the major tick mark spacing at 10. As a result, major tick marks will be displayed at the values 0, 10, 20, 30, 40, and 50.
- Line 4 causes 5 minor tick marks to be displayed between the major tick marks.
- Line 5 displays numeric labels at the major tick marks.
- Line 6 sets the width of the `Slider` to 300 pixels.

The following code snippet creates the vertical `Slider` shown in Figure 13-26:

```
1   Slider vSlider = new Slider(-1.0, 1.0, 0.0);
2   vSlider.setOrientation(Orientation.VERTICAL);
3   vSlider.setShowTickMarks(true);
4   vSlider.setMajorTickUnit(0.25);
5   vSlider.setMinorTickCount(5);
6   vSlider.setShowTickLabels(true);
7   vSlider.setPrefHeight(300.0);
```

Let's take a closer look at the code:

- Line 1 creates a `Slider` with a minimum value of −1, and a maximum value of 1. The `Slider`'s initial value is 0.
- Line 2 sets the `Slider`'s orientation to vertical.
- Line 3 causes the `Slider` to display tick marks.
- Line 4 sets the major tick mark spacing at 0.25. As a result, major tick marks will be displayed at the values −1, −0.75, −0.5, −0.25, 0, 0.25, 0.5, 0.75, and 1.
- Line 5 causes 5 minor tick marks to be displayed between the major tick marks.
- Line 6 displays numeric labels at the major tick marks.
- Line 7 sets the width of the `Slider` to 300 pixels.

When a `Slider`'s value changes (because the knob was moved), the `Slider` control generates a *change event*. If you want an action to take place when the `Slider`'s value changes, you can register an event handler for the change event. Assuming `slider` is the name of a `Slider` control, the following code shows an example of how to register such an event handler with a lambda expression:

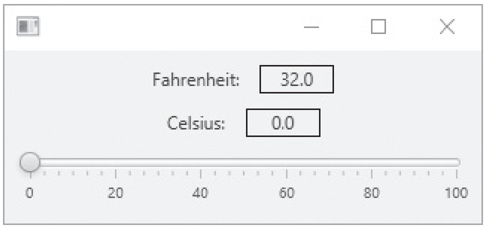
```
slider.valueProperty().addListener(
   (observeable, oldvalue, newvalue) ->
   {
      //  Write event handling code here...
   });
```

Code Listing 13-25 demonstrates the `Slider` control. This program displays the window shown in Figure 13-27. Two temperatures are initially shown: 32.0 degrees Fahrenheit and 0.0 degrees Celsius. A `Slider`, which has the range of 0 through 100, allows you to adjust the Celsius temperature and immediately see the Fahrenheit conversion.

**Code Listing 13-25** `(SliderDemo.java)`

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.geometry.Pos;
import javafx.geometry.Insets;

public class SliderDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Slider constants
      final double MIN = 0.0, MAX = 100.0, INITIAL = 0.0;
      final double MAJOR_TICK_UNIT = 20.0;
      final int MINOR_TICK_COUNT = 5;
      final double SLIDER_WIDTH = 300.0;

      // Miscellaneous constants
      final double LABEL_WIDTH = 50.0;
      final double SPACING = 10.0;

      // Create the Fahrenheit controls.
      Label fDescriptor = new Label("Fahrenheit: ");
      Label fLabel = new Label("32.0");
      fLabel.setStyle("-fx-border-style: solid;" +
                      "-fx-alignment: center");
      fLabel.setPrefWidth(LABEL_WIDTH);
      HBox fHBox = new HBox(SPACING, fDescriptor, fLabel);
      fHBox.setAlignment(Pos.CENTER);

      // Create the Celsius controls.
      Label cDescriptor = new Label("Celsius: ");
      Label cLabel = new Label("0.0");
      cLabel.setStyle("-fx-border-style: solid;" +
                      "-fx-alignment: center");
      cLabel.setPrefWidth(LABEL_WIDTH);
      HBox cHBox = new HBox(SPACING, cDescriptor, cLabel);
      cHBox.setAlignment(Pos.CENTER);

      // Make the Slider.
      Slider slider = new Slider(MIN, MAX, INITIAL);
      slider.setShowTickMarks(true);
      slider.setMajorTickUnit(MAJOR_TICK_UNIT);
      slider.setMinorTickCount(MINOR_TICK_COUNT);
      slider.setShowTickLabels(true);
      slider.setSnapToTicks(true);
      slider.setPrefWidth(SLIDER_WIDTH);

      // Register an event handler for the Slider.
      slider.valueProperty().addListener(
         (observeable, oldvalue, newvalue) ->
         {
            // Get the Celsius temp from the Slider.
            double celsius = slider.getValue();

            // Convert Celsius to Fahrenheit.
            double fahrenheit = (9.0 / 5.0) * celsius + 32;

            // Display the Celsius and Fahrenheit temps.
            cLabel.setText(String.format("%.1f", celsius));
            fLabel.setText(String.format("%.1f", fahrenheit));
         });

      // Add the controls to an VBox.
      VBox vbox = new VBox(10, fHBox, cHBox, slider);
      vbox.setAlignment(Pos.CENTER);
      vbox.setPadding(new Insets(SPACING));

      // Create a Scene and display it.
      Scene scene = new Scene(vbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 13-27** The SliderDemo.java application (Oracle Corporate Counsel)



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

13.28 What `Slider` methods do you use to perform each of these operations?

- a) Establish the spacing of major tick marks.
- b) Establish the spacing of minor tick marks.
- c) Cause tick marks to be displayed.
- d) Cause labels to be displayed.

13.29 What type of event does a `Slider` generate when its slider knob is moved?

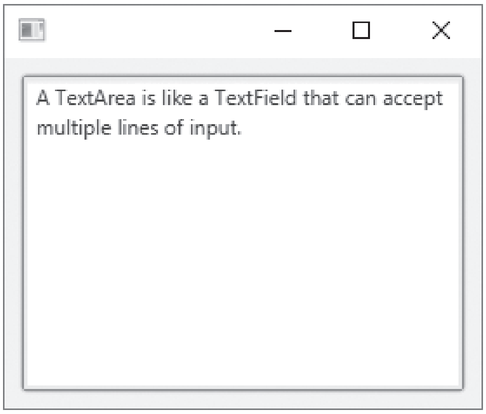
13.30 How do you get a `Slider`'s current value?

## 13.7 TextArea Controls

**CONCEPT: A `TextArea` is a multiline `TextField` that can accept several lines of text input, or display several lines of text.**

In Chapter 12, you were introduced to the `TextField` control, which allows the user to enter a single line of text. A `TextArea` is like a `TextField` that can accept multiple lines of input. Figure 13-28 shows an example.

**Figure 13-28** The TextArea control (Oracle Corporate Counsel)



A TextArea is like a TextField that can accept multiple lines of input.

The `TextArea` class, which is in the `javafx.scene.control` package, has two constructors. The no-arg constructor creates an empty `TextArea`. Here is a statement that creates a `TextArea` using the no-arg constructor:

```
TextArea textArea = new TextArea();
```

The second constructor accepts a `String` argument, as shown in this example:

```
TextArea textArea = new TextArea("This is the initial text.");
```

The string that is passed to the `TextArea` constructor will be initially displayed in the `TextArea`. Table 13-8 lists some of the methods you will use most often.

**Table 13-8** Some of the `TextArea` class methods

| Method | Description |
|---|---|
| `setPrefColumnCount(`*`value`*`)` | The *`value`* argument is an `int`. Sets the preferred number of text columns. (The default value is 40.) |
| `setPrefRowCount(`*`value`*`)` | The *`value`* argument is an `int`. Sets the preferred number of text rows. (The default value is 10.) |
| `setWrapText(`*`value`*`)` | The *`value`* argument is a `boolean`. If the argument is `true`, then the `TextArea` will wrap a line of text that exceeds the end of a row. If the argument is `false`, then the `TextArea` will scroll when a line of text exceeds the end of a row. (The default value is `false`.) |
| `getText()` | Returns the contents of the `TextArea` as a `String`. |
| `setText(`*`value`*`)` | The *`value`* argument is a `String`. The argument becomes the text that is displayed in the `TextArea` control. |

The following code shows an example of creating an empty `TextArea` control with a width of 20 columns and a height of 30 rows:

```
TextArea textArea = new TextArea();
textArea.setPrefColumnCount(20);
textArea.setPrefRowCount(30);
```

By default, `TextArea` controls do not perform *text wrapping*. This means when text is entered into the control and the end of a line is reached, the text does not wrap around to the next line. Instead, the `TextArea` scrolls to the right, and displays a horizontal scrollbar.

As an alternative, you can enable text wrapping with the `TextArea` class's `setWrapText` method. The method accepts a `boolean` argument. If you pass `true`, then text wrapping is enabled. If you pass `false`, text wrapping is disabled. Here is an example:

```
textInput.setWrapText(true);
```

With line wrapping enabled, a line of text that exceeds the end of a row will be wrapped to the next line.

**NOTE:** The `TextArea` control performs a type of wrapping known as *word wrapping*. This means the line breaks always occur between words, not in the middle of a word.

To retrieve the text the user has typed into a `TextArea` control, you call the control's `getText` method. The method returns the text that has been entered into the `TextField` as a `String`. For example, assume `textArea` is a `TextArea` control. The following code reads the text that has been entered into `textArea`, and assigns it to a `String` variable named `input`:

```
String input;
input = textArea.getText();
```

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.31 What is the difference between a `TextArea` and a `TextField`?

13.32 What is the `TextArea`'s default number of columns displayed? How do you change the number of columns?

13.33 What is the `TextArea`'s default number of rows displayed? How do you change the number of rows?

13.34 What is text wrapping? What is line wrapping? How do you enable text wrapping in a `TextArea` control?

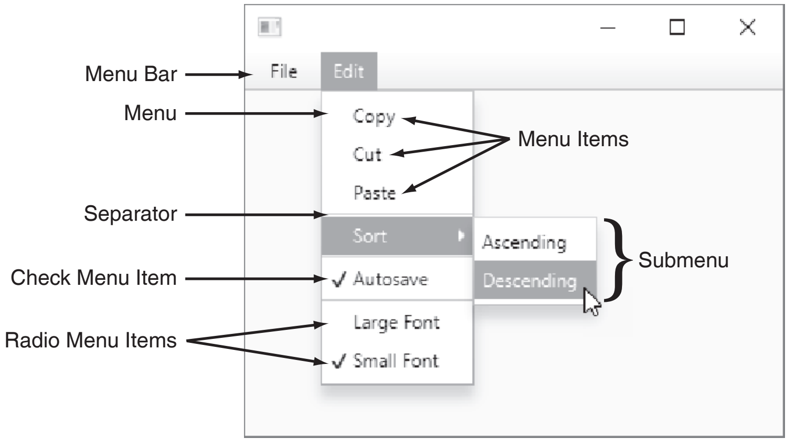
13.35 How do you retrieve the text that the user has typed into a `TextArea` control?

## 13.8 Menus

**CONCEPT: JavaFX provides classes for creating systems of drop-down menus. Menus can contain menu items, checked menu items, radio button menu items, and other menus.**

In the GUI applications you have studied so far, the user initiates actions by clicking on controls such as `Button`s. When an application has several operations for the user to choose from, a menu system is more commonly used than buttons. A *menu system* is a collection of commands organized in one or more drop-down menus. Before learning how to construct a menu system, you must learn about the basic items found in a typical menu system. Look at the example menu system in Figure 13-29.

**Figure 13-29** Example menu system (Oracle Corporate Counsel)



The menu system in the figure consists of the following items:

- **Menu bar.** At the top of the window is a menu bar. The *menu bar* lists the names of one or more menus. The menu bar in Figure 13-29 shows the names of two menus: *File* and *Edit*.
- **Menu.** A *menu* is a drop-down list of menu items. The user may activate a menu by clicking on its name on the menu bar. In Figure 13-29, the *Edit* menu has been activated.
- **Menu Item.** A *menu item* can be selected by the user. When a menu item is selected, some type of action is usually performed.
- **Check menu item.** A *check menu item* appears that may be selected or deselected. When it is selected, a check mark appears next to the item. When it is deselected, the check does not appear. Check menu items are normally used to turn an option on or off. The user toggles the state of a check menu item each time he or she selects it.
- **Radio menu item.** A *radio menu item* may be selected or deselected. A radio menu item looks like a check menu item. When it is selected, a check mark appears next to the item. When it is deselected, the check does not appear. When a set of radio menu items are grouped with a `ToggleGroup` object, only one of them can be selected at a time. When the user selects a radio menu item, the one that was previously selected is deselected.
- **Submenu.** A menu within a menu is called a *submenu*. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear. For example, in Figure 13-29, clicking on the *Sort* command causes a submenu to appear.
- **Separator bar.** A separator bar is a horizontal bar used to separate groups of items on a menu. Separator bars are only used as a visual aid and cannot be selected by the user.

A menu system is constructed with the following classes:

**`MenuBar`**. Use this class to create a menu bar. A `MenuBar` object can contain `Menu` objects.

**`Menu`**. Use this class to create a menu that drops down from the menu bar. A `Menu` object can contain `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` objects, as well as other `Menu` objects. A submenu is a `Menu` object that is inside another `Menu` object.

**`MenuItem`**. Use this class to create a regular menu item. A `MenuItem` object generates an action event when the user selects it.

**`CheckMenuItem`**. Use this class to create a check menu item. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `CheckMenuItem` object generates an action event when the user selects it.

**`RadioMenuItem`**. Use this class to create a radio button menu item. `RadioMenuItem` objects can be grouped together in a `ToggleGroup` object so only one of them can be selected at a time. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `RadioMenuItem` object generates an action event when the user selects it.

All of these classes are in the `javafx.scene.control` package. A menu system is a `MenuBar` that contains one or more `Menu` objects. Each `Menu` object can contain `MenuItem`, `RadioMenuItem`, and `CheckMenuItem` objects, as well as other `Menu` objects. The process of building a menu system requires the following steps (although not necessarily in this order):

- Create a `MenuBar` object. This will be the "root" of the menu system.
- Create the necessary `Menu` objects, and add them to the `MenuBar` object.
- For each `Menu` object, create the necessary menu item objects, and add them to their `Menu` object.
- Register an event handler for each menu item object.
- Add the `MenuBar` object to the scene graph.

To see an example of an application that uses a simple menu system, look at Code Listing 13-26. The program creates a menu bar with one menu: *File*. The *File* menu has one menu item: *Exit*. When the user selects the *Exit* menu item, the program stops. Figure 13-30 shows the program's menu system.

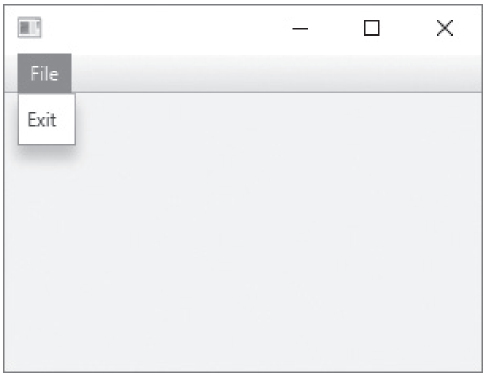
**Code Listing 13-26** (`SimpleMenu.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.MenuBar;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.BorderPane;

public class SimpleMenu extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene dimensions
      final double WIDTH = 300.0, HEIGHT = 200.0;

      // Create the menu bar.
      MenuBar menuBar = new MenuBar();

      // Create the File menu.
      Menu fileMenu = new Menu("File");
      MenuItem exitItem = new MenuItem("Exit");
      fileMenu.getItems().add(exitItem);

31          // Register an event handler for the exit item.
32          exitItem.setOnAction(event ->
33          {
34             primaryStage.close();
35          });
36
37          // Add the File menu to the menu bar.
38          menuBar.getMenus().add(fileMenu);
39
40          // Add the menu bar to a BorderPane.
41          BorderPane borderPane = new BorderPane();
42          borderPane.setTop(menuBar);
43
44          // Create a Scene and display it.
45          Scene scene = new Scene(borderPane, WIDTH, HEIGHT);
46          primaryStage.setScene(scene);
47          primaryStage.show();
48       }
49    }
```

Let's take a closer look at the program:

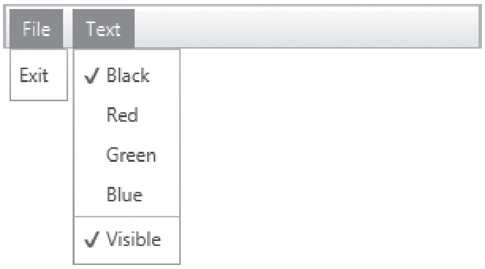
- Line 21 declares constants for the scene dimensions.
- Line 24 creates a `MenuBar` object.
- Line 27 creates a `Menu` object for the *File* menu.
- Line 28 creates a `MenuItem` object for the *Exit* menu item.
- Line 29 adds the *Exit* menu item to the *File* menu.
- Lines 32 through 35 register an event handler for the *Exit* menu item. When the user clicks the *Exit* menu item, line 34 closes the program.
- Line 38 adds the *File* menu to the menu bar.
- Lines 41 and 42 create a `BorderPane` layout container and add the menu bar to the top region.
- Lines 45 through 47 create the scene, set the scene to the stage, and display the application's window.

**Figure 13-30** The SimpleMenu.java application (Oracle Corporate Counsel)



Code Listing 13-27 is an application with a slightly more complex menu system, which is shown in Figure 13-31. This application demonstrates how the text displayed by a `Label` control appears in different colors.

**Figure 13-31** Menu system in the TextMenu.java program (Oracle Corporate Counsel)



The menu bar has two menus: *File* and *Text*. When the user opens the *File* menu, and selects the *Exit* item, the application ends. When the user opens the *Text* menu, he or she can select a color using the *Black*, *Red*, *Green*, and *Blue* radio menu items. These items change the color of the "Hello World" text that is displayed in a `Label` control. The *Text* menu also has a *Visible* item, which is a check menu item. When this item is selected (checked), the `Label` control is made visible. When this item is deselected (unchecked), the `Label` control is made invisible. The application's window is shown in Figure 13-32.

**Code Listing 13-27** (`TextMenu.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.MenuBar;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuItem;
import javafx.scene.control.CheckMenuItem;
import javafx.scene.control.RadioMenuItem;
import javafx.scene.control.SeparatorMenuItem;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.BorderPane;

public class TextMenu extends Application
{
   // Fields for the menu components
   private MenuBar menuBar;
   private Menu fileMenu;
   private Menu textMenu;
   private MenuItem exitItem;
   private RadioMenuItem blackItem;
   private RadioMenuItem redItem;
   private RadioMenuItem greenItem;
   private RadioMenuItem blueItem;
   private CheckMenuItem visibleItem;

   // Field for the text
   private Label outputLabel;

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene dimensions
      final double WIDTH = 300.0, HEIGHT = 200.0;

      // Create the Label control.
      outputLabel = new Label("Hello World");

      // Create the menu bar.
      menuBar = new MenuBar();

      // Create the File menu.
      buildFileMenu(primaryStage);

      // Create the Text menu.
      buildTextMenu();

      // Add the File & Text menus to the menu bar.
      menuBar.getMenus().add(fileMenu);
      menuBar.getMenus().add(textMenu);

      // Add the controls to a BorderPane.
      BorderPane borderPane = new BorderPane();
      borderPane.setTop(menuBar);
      borderPane.setCenter(outputLabel);

      // Create a Scene and display it.
      Scene scene = new Scene(borderPane, WIDTH, HEIGHT);
      primaryStage.setScene(scene);
      primaryStage.show();
   }

   /**
     This method builds the File menu.
   */

   private void buildFileMenu(Stage primaryStage)
   {
      // Create the File Menu object.
      fileMenu = new Menu("File");

      // Create the Exit MenuItem object.
      exitItem = new MenuItem("Exit");

      // Register an event handler for the Exit item.
      exitItem.setOnAction(event ->
      {
         primaryStage.close();
      });

      // Add the Exit item to the File menu.
      fileMenu.getItems().add(exitItem);
    }

    /**
      This method builds the Text menu.
    */

    private void buildTextMenu()
    {
       // Create the Text Menu object.
       textMenu = new Menu("Text");

      // Create the menu items for the Text menu.
      blackItem = new RadioMenuItem("Black");
      redItem = new RadioMenuItem("Red");
      greenItem = new RadioMenuItem("Green");
      blueItem = new RadioMenuItem("Blue");
      visibleItem = new CheckMenuItem("Visible");

      // Select the Black and Visible menu items.
      blackItem.setSelected(true);
      visibleItem.setSelected(true);

      // Add the RadioMenuItems to a ToggleGroup.
      ToggleGroup textToggleGroup = new ToggleGroup();
      blackItem.setToggleGroup(textToggleGroup);
      redItem.setToggleGroup(textToggleGroup);
      greenItem.setToggleGroup(textToggleGroup);
      blueItem.setToggleGroup(textToggleGroup);

```

```
118       // Register event handlers for the menu items.
119       blackItem.setOnAction(event ->
120      {
121         outputLabel.setStyle("-fx-text-fill: black");
122      });
123
124      redItem.setOnAction(event ->
125      {
126         outputLabel.setStyle("-fx-text-fill: red");
127      });
128
129      greenItem.setOnAction(event ->
130      {
131         outputLabel.setStyle("-fx-text-fill: green");
132      });
133
134      blueItem.setOnAction(event ->
135      {
136         outputLabel.setStyle("-fx-text-fill: blue");
137      });
138
139      visibleItem.setOnAction(event ->
140      {
141         if (outputLabel.isVisible())
142            outputLabel.setVisible(false);
143         else
144            outputLabel.setVisible(true);
145      });
146
147      // Add the menu items to the Text menu.
148      textMenu.getItems().add(blackItem);
149      textMenu.getItems().add(redItem);
150      textMenu.getItems().add(greenItem);
151      textMenu.getItems().add(blueItem);
152      textMenu.getItems().add(new SeparatorMenuItem());
153      textMenu.getItems().add(visibleItem);
154    }
155 }
```

Lines 17 through 25 declare private fields for the menu components. We declared these fields because the code that builds the menu system is modularized into methods, and we need to access these objects from the different methods. Line 28 declares `outputLabel` as a private field for the same reason.

Let's take a closer look at the `start` method, in lines 37 through 67:

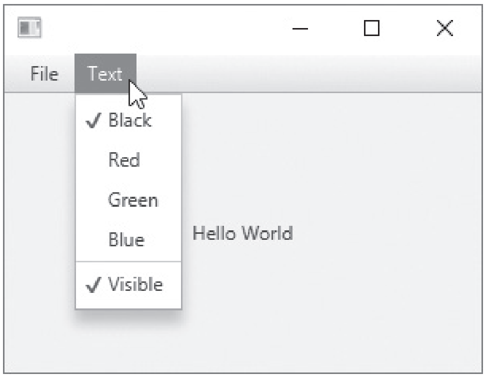
- Line 40 declares constants for the scene dimensions.
- Line 43 creates the `outputLabel` control.
- Line 46 creates the `MenuBar`.

- Line 49 calls the `buildFileMenu` method, which builds the *File* menu. Notice we are passing the `primaryStage` variable as an argument. Recall that the `primaryStage` variable references the application's `Stage`. The `buildFileMenu` method needs to access the `Stage` because the statement in line 84 calls the `Stage` object's `close()` method to end the application.
- Line 52 calls the `buildTextMenu` method, which builds the *Text* menu.
- Lines 55 and 56 add the *File* menu and the *Text* menu to the menu bar.
- Lines 59 through 61 create a `BorderPane`, put the menu bar in the top region, and place the `Label` control in the center region.
- Lines 64 through 66 create the scene, set the scene to the stage, and display the application's window.

The `buildFileMenu` method appears in lines 73 through 89. When this method is finished, the *File* menu will be constructed, and an event handler will be registered for that menu's *Exit* item.

The `buildTextMenu` method appears in lines 95 through 154. When this method is finished, the *Text* menu will be constructed, and event handlers will be registered for all of that menu's items.

**Figure 13-32** The TextMenu.java application (Oracle Corporate Counsel)



## Assigning Mnemonics to Menu Items

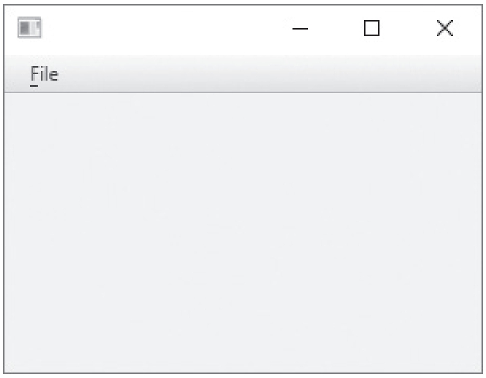
A *mnemonic* is a key on the keyboard that you press in combination with the Alt key to access a menu item quickly. Mnemonics are sometimes referred to as *shortcut keys*, *access keys*, or *hot keys*. You assign a mnemonic to a menu item when you instantiate the menu item. The following code snippet shows an example:

```
Menu fileMenu = new Menu("_File");
MenuItem exitItem = new MenuItem("E_xit");
```

Notice the string we are passing to the `Menu` class constructor in the first statement. An underscore character appears immediately before the letter `'F'`. This will cause the letter F to become a mnemonic. When the user presses Alt+F on the keyboard, the *File* menu will be automatically selected. Also notice the string that we are passing to the `MenuItem` class constructor in the second statement. An underscore character appears immediately before the letter `'x'`. This will cause the letter X to become a mnemonic. When the user presses Alt+X on the keyboard, the *Exit* menu item will be selected.

The underscore character will not be displayed in the name of the menu item at runtime. However, when the user presses the Alt key on the keyboard, the mnemonic character will appear underlined, as shown in Figure 13-33.

**Figure 13-33** The *File* menu with the Alt+F mnemonic (Oracle Corporate Counsel)



**NOTE:** Mnemonics do not distinguish between uppercase and lowercase characters. There is no difference between Alt+X and Alt+x.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.36 Briefly describe each of the following menu system items:
- a) Menu bar
- b) Menu
- c) Menu item
- d) Check menu item
- e) Radio menu item
- f) Submenu
- g) Separator bar

13.37 What class do you use to create a menu bar?

13.38 What class do you use to create a menu?

13.39 What class do you use to create a menu item?

13.40 What class do you use to create a radio menu item? How do you cause it to be initially selected?

13.41 How do you create a relationship between radio menu items so only one may be selected at a time?

13.42 What class do you use to create a check menu item? How do you cause it to be initially selected?

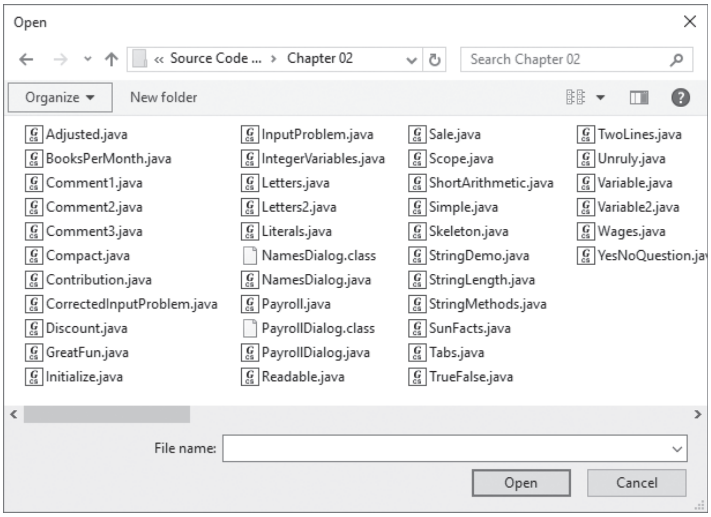
13.43 What type of event do menu items generate when selected by the user?

## 13.9 The `FileChooser` Class

**CONCEPT:** **JavaFX provides a `FileChooser` class that equips your applications with standard dialog boxes for opening files and saving files.**

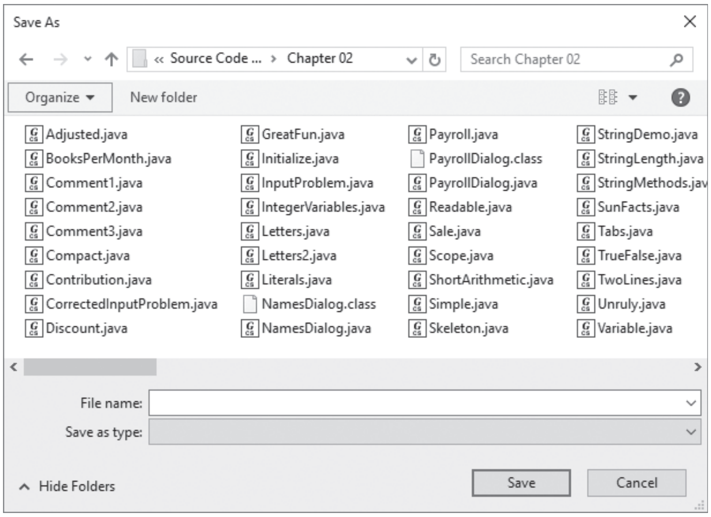
The `FileChooser` class displays a specialized dialog box that allows the user to browse for a file and select it. Figure 13-34 shows an example of a file chooser dialog box on a Windows system.

**Figure 13-34** A file chooser dialog box for opening a file (Oracle Corporate Counsel)


Open
« Source Code ... › Chapter 02
Search Chapter 02
Organize ▾
New folder
Adjusted.java
BooksPerMonth.java
Comment1.java
Comment2.java
Comment3.java
Compact.java
Contribution.java
CorrectedInputProblem.java
Discount.java
GreatFun.java
Initialize.java
InputProblem.java
IntegerVariables.java
Letters.java
Letters2.java
Literals.java
NamesDialog.class
NamesDialog.java
Payroll.java
PayrollDialog.class
PayrollDialog.java
Readable.java
Sale.java
Scope.java
ShortArithmetic.java
Simple.java
Skeleton.java
StringDemo.java
StringLength.java
StringMethods.java
SunFacts.java
Tabs.java
TrueFalse.java
TwoLines.java
Unruly.java
Variable.java
Variable2.java
Wages.java
YesNoQuestion.ja
File name:
Open
Cancel

The `FileChooser` class is in the `javafx.stage` package. The `FileChooser` class can display two types of predefined dialog boxes: an *open dialog box* and a *save dialog box*. The dialog box shown in Figure 13-34 is an open dialog box. This dialog box lets the user browse for an existing

**Figure 13-35** A file chooser dialog box for saving a file (Oracle Corporate Counsel)


Save As
« Source Code ... › Chapter 02
Search Chapter 02
Organize ▾
New folder
Adjusted.java
BooksPerMonth.java
Comment1.java
Comment2.java
Comment3.java
Compact.java
Contribution.java
CorrectedInputProblem.java
Discount.java
GreatFun.java
Initialize.java
InputProblem.java
IntegerVariables.java
Letters.java
Letters2.java
Literals.java
NamesDialog.class
NamesDialog.java
Payroll.java
PayrollDialog.class
PayrollDialog.java
Readable.java
Sale.java
Scope.java
ShortArithmetic.java
Simple.java
Skeleton.java
StringDemo.java
StringLength.java
StringMethods.jav
SunFacts.java
Tabs.java
TrueFalse.java
TwoLines.java
Unruly.java
Variable.java
File name:
Save as type:
Hide Folders
Save
Cancel

file to open. A save dialog box, as shown in Figure 13-35, is employed when the user needs to browse to a location to save a file. Both of these dialog boxes are similar in the way they operate.

## Displaying a `FileChooser` Dialog Box

To display either an open dialog box or a save dialog box, you first create an instance of the `FileChooser` class. Then, you call either the `showOpenDialog` method (if you want to display an open dialog box), or the `showSaveDialog` method (if you want to display a save dialog). With either method, you pass a reference to the application's stage as an argument. The following code snippet shows an example of calling the `showOpenDialog` method:

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(primaryStage);
```

In this code, assume `primaryStage` references our application's stage. The `showOpenDialog` and `showSaveDialog` methods return a `File` object (the `File` class, which was introduced in Chapter 4, is in the `java.io` package) containing information about the selected file. If the user does not select a file, the method returns `null`. You can typically use the `File` object to open the file, or create a file. The following code snippet shows how you can use the `File` object's `getPath` method to get the path and file name as a `String`:

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(primaryStage);
if (selectedFile != null)
{
   String filename = selectedFile.getPath();
   outputLabel.setText("You selected " + filename);
}
```

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.44 In what package is the `FileChooser` class?

13.45 How do you display an open dialog?

13.46 How do you display a save dialog?

13.47 How do you determine the file that the user selected with either an open dialog or a save dialog?

## 13.10 Using Console Output to Debug a GUI Application

**CONCEPT: When debugging a GUI application, you can use `System.out.print`, `System.out.println`, or `System.out.printf` to send diagnostic messages to the console.**

When an application is not performing correctly, programmers sometimes write statements that display *diagnostic messages* into the application. For example, if an application is not giving the correct result for a calculation, diagnostic messages can be displayed at various points in the program's execution showing the values of all the variables used in the calculation. If the trouble is caused by a variable that has not been properly initialized, or that has not been assigned the correct value, the diagnostic messages reveal this problem. This helps the programmer to see what is going on "under the hood" while an application is running.

The console output methods, `System.out.print`, `System.out.println` and `System.out.printf`, can be valuable tools for displaying diagnostic messages in a GUI application. Because these methods send their output to the console, diagnostic messages can be displayed without interfering with the application's GUI windows. Code Listing 13-28 shows an example. This is a modified version of the LambdaKiloConverter program that you saw in Chapter 12. In this version of the program, we have inserted statements that write console messages inside the `calcButton`'s event handler. The lines containing the calls are shown in bold. These calls display the value that the application has retrieved from the text field, and is working within its calculation.

**Code Listing 13-28** (`ConsoleDebugging.java`)

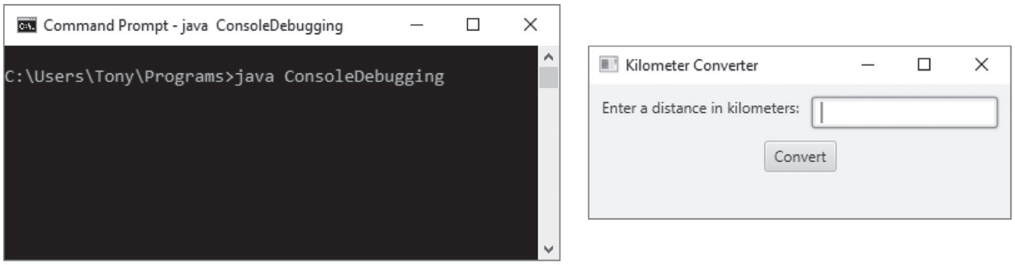
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
   Kilometer Converter application with console debugging
*/

public class ConsoleDebugging extends Application
{
   // Fields
   private TextField kiloTextField;
   private Label resultLabel;

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Create a Label to display a prompt.
      Label promptLabel = new Label("Enter a distance in kilometers:");

      // Create a TextField for input.
      kiloTextField = new TextField();

      // Create a Button to perform the conversion.
      Button calcButton = new Button("Convert");

      // Create an event handler.
      calcButton.setOnAction(event ->
      {
         // For debugging, display the text entered, and
         // its value converted to a double.
         System.out.printf("Reading %s from the text field. \n",
                           kiloTextField.getText());
         System.out.printf("Converted value: %f\n",
                           Double.parseDouble(kiloTextField.getText()));

         // Get the kilometers.
         Double kilometers = Double.parseDouble(kiloTextField.getText());

         // Convert the kilometers to miles.
         Double miles = kilometers * 0.6214;

         // Display the results.
         resultLabel.setText(String.format("%,.2f miles", miles));

         // For debugging, display a message indicating
         // the application is ready for more input.
         System.out.println("Ready for the next input.");
      });

      // Create an empty Label to display the result.
      resultLabel = new Label();

      // Put the promptLabel and the kiloTextField in an HBox.
      HBox hbox = new HBox(10, promptLabel, kiloTextField);

      // Put the HBox, calcButton, and resultLabel in a VBox.
      VBox vbox = new VBox(10, hbox, calcButton, resultLabel);

      // Set the VBox's alignment to center.
      vbox.setAlignment(Pos.CENTER);

      // Set the VBox's padding to 10 pixels.
      vbox.setPadding(new Insets(10));

```

```
81        // Create a Scene.
82        Scene scene = new Scene(vbox);
83
84        // Add the Scene to the Stage.
85        primaryStage.setScene(scene);
86
87        // Set the stage title.
88        primaryStage.setTitle("Kilometer Converter");
89
90        // Show the window.
91        primaryStage.show();
92     }
93  }
```

Let's take a closer look. In lines 47 and 48, a message is displayed to the console showing the value that was read from the text field. In lines 49 and 50, another message is displayed showing the value after it is converted to a double. Then, in line 63, a message is displayed indicating the application is ready for its next input. Figure 13-36 shows an example session with the application on a computer running Windows. Both the console window and the application windows are shown.

**Figure 13-36** Messages displayed to the console during the application's execution (Oracle Corporate Counsel)

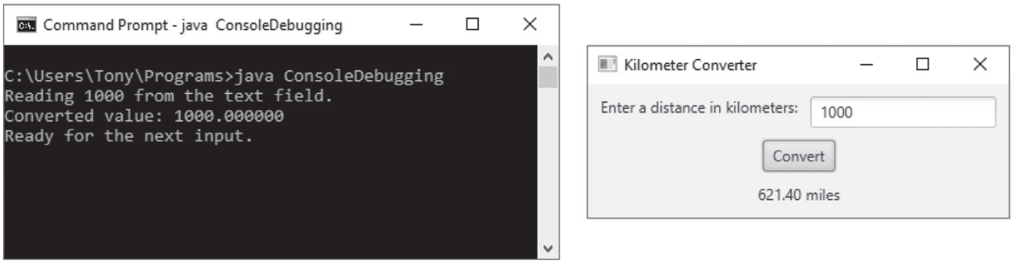
1. A command is typed in the console window to execute the application. The application's window appears.



```
C:\Users\Tony\Programs>java ConsoleDebugging
```

Enter a distance in kilometers:

2. The user types a value into the text field and clicks the Convert button. Debugging messages appear in the console window.



```
C:\Users\Tony\Programs>java ConsoleDebugging
Reading 1000 from the text field.
Converted value: 1000.000000
Ready for the next input.
```

Enter a distance in kilometers: 1000

621.40 miles

The messages displayed to the console are meant only for the programmer to see while he or she is debugging the application. Once the programmer is satisfied that the application is running correctly, the statements that display the messages can be taken out.

## 13.11 Common Errors to Avoid

- **Forgetting to add `RadioButton` controls to a `ToggleGroup`.** A mutually exclusive relationship is created between `RadioButton` controls only when they are added to a `ToggleGroup` object.
- **Forgetting to initially select one of the `RadioButton` controls in a `ToggleGroup`.** In most situations, you want one of the `RadioButton` controls in a `ToggleGroup` to be initially selected. In code, you can select a `RadioButton` control by calling its `setSelected` method, passing `true` as the argument.
- **Adding items to a `ListView` when you really meant to set items, or vice versa.** The `ListView` control's `getItems().addAll()` method adds items to the `ListView`. If the `ListView` already contains items, the `getItems().addAll()`method will not erase the existing items, but will add the new items to the existing list. The `getItems().setAll()` method, on the other hand, replaces a `ListView` control's existing items with a new list of items.
- **Forgetting to add menu items to a `Menu` control, and a `Menu` control to a `MenuBar` control.** After you create a menu item, you must add it to a `Menu` control in order for it to be displayed on the menu. Likewise, `Menu` controls must be added to a `MenuBar` control in order to be displayed on the `MenuBar`.
- **Not grouping `RadioMenuItem` controls in a `ToggleGroup` object.** Just like regular `RadioButton` controls, you must group `RadioMenuItem` controls in a `ToggleGroup` in order to create a mutually exclusive relationship between them.

## Review Questions and Exercises

### Multiple Choice and True/False

1. When a selector name starts with a period in a JavaFX CSS style definition, it means the selector corresponds to __________.
    a. specific variable names in the application
    b. specific types of JavaFX nodes
    c. values that the user enters
    d. named constants in the application
2. Which of these is written in the correct syntax for a CSS style definition?
    a. `{.label: -fx-font-size: 20pt; }`
    b. `.label { -fx-font-size=20pt; }`
    c. `.label { -fx-font-size: 20pt; }`
    d. `{.label; -fx-font-size: 20pt; }`
3. If you want to apply styles to all of the nodes in a scene, use the __________ selector.
    a. `.all-nodes`
    b. `.stage`
    c. `.home`
    d. `.root`

4. The __________ CSS property is used to change a control's background color in a JavaFX application.
   a. `-fx-color`
   b. `-fx-background-color`
   c. `-fx-color-background`
   d. `-fx-bg-color`
5. When you define a color in CSS, you specify an integer value in the range of __________ for each of the color components (red, green, and blue).
   a. 0 through 255
   b. 0 through 1024
   c. −255 through 255
   d. 0 through 128
6. In the hexadecimal color value `#05AAFF`, the `AA` value specifies which color component?
   a. red
   b. green
   c. blue
   d. magenta
7. What style definition applies a background color of `#0000FF` to all `Button` controls?
   a. `.button { -fx-background-color: #0000FF; }`
   b. `.button { -fx-bg-color = #0000FF; }`
   c. `.button.all { -fx-background-color: #0000FF; }`
   d. `.global {button: -fx-background-color: #0000FF; }`
8. Adding `RadioButton` controls to this type of object creates a mutually exclusive relationship between them.
   a. `MutualExclude`
   b. `RadioGroup`
   c. `LogicalGroup`
   d. `ToggleGroup`
9. This `RadioButton` method returns `true` if this control is selected.
   a. `selected()`
   b. `clicked()`
   c. `isSelected()`
   d. `isChecked()`
10. The __________ control displays a list of items and allows the user to select one or more items from the list.
   a. `DropList`
   b. `ListView`
   c. `ViewList`
   d. `ItemList`
11. The __________ control presents its items in a drop-down list.
   a. `DropList`
   b. `ListView`
   c. `ComboBox`
   d. `ItemList`

12. Both the `ListView` control and the `ComboBox` control keeps its list of items in a(n) __________ object.
    a. `ArrayList`
    b. `ExpandableList`
    c. `ItemList`
    d. `ObservableList`
13. A __________ is like a `TextField` that can accept multiple lines of input.
    a. `TextArea`
    b. `MultilineTextField`
    c. `ExpandableTextField`
    d. `TextEditorField`
14. You use this class to create a menu bar.
    a. `MenuItem`
    b. `MenuBar`
    c. `Menu`
    d. `Bar`
15. You use this class to create a radio menu item.
    a. `MenuItem`
    b. `RadioButton`
    c. `RadioItem`
    d. `RadioMenuItem`
16. **True or False:** If you make any changes to an application's stylesheet, you have to recompile the Java application in order for the style changes to take effect.
17. **True or False:** If a style definition for a specific type of node contradicts a style rule in the `.root` definition, the more specific style definition takes precedence over the `.root` definition.
18. **True or False:** You can list only one selector in the first line of a style definition.
19. **True or False:** A node's `setStyle` method removes any styles that were previously applied to the node, and replaces them with the new style rule that is passed as an argument.
20. **True or False:** A scroll bar automatically appears when a `ListView` contains more items than can be displayed in the space provided.
21. **True or False:** By default, `ListView` controls are oriented horizontally.
22. **True or False:** By default, `TextArea` controls perform line wrapping.
23. **True or False:** A `MenuBar` object acts as a container for `Menu` objects.
24. **True or False:** A `Menu` object cannot contain other `Menu` objects.

## Find the Error

1. `.button { -fx-background-color = #0000FF; }`
2. `.label { -font-size: 14pt; }`
3. 
```
// This code has an error!
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
ToggleGroup radioGroup = new ToggleGroup();
```

```
radioGroup.setToggleGroup(radio1);
radioGroup.setToggleGroup(radio2);
```

4.
```
// This code has an error!
ListView<String> myListView = new ListView<>();
myListView.getItems().addAll(10, 20, 30, 40);
```

## Algorithm Workbench

1. Write a JavaFX CSS style definition that will be applied to all `Label` controls. The style definition should specify the following: font size: 24, font style: italic, font weight: bold.
2. Suppose we have a stylesheet named *styles.css*, and in our Java code the `scene` variable references the `Scene` object. Write a statement to add the stylesheet to the scene.
3. Write a JavaFX CSS style definition that will be applied to all `Label`, `Button`, and `TextField` controls. The style definition should specify the following: font size: 24, font style: italic, font weight: bold.
4. Assume `scene` is the name of a `Scene` object, and *styles.css* is the name of a CSS stylesheet. Write the statement that applies the *styles.css* stylesheet to `scene`.
5. Write functional RGB notation for a color where red = 100, green = 20, and blue = 255.
6. An application has a `Button` control named `calcButton`, and you have applied a stylesheet to the root node. In that style sheet, you have defined a custom style class named `button-red`. Write the statement that would apply the `button-red` custom style class to the `calcButton` control.
7. An application has a `Button` control named `calcButton`. Write the statement that assigns the ID `"calc-button"` to the `calcButton` control.
8. An application has a `Label` control named `myLabel`. Write a statement that applies an inline style rule to `myLabel`. The style rule should set the font size to 12 points.
9. Write the code to add the following `RadioButton` controls to the `ToggleGroup`.

```
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
RadioButton radio3 = new RadioButton("Option 3");
ToggleGroup radioGroup = new ToggleGroup();
```

10. Assume a JavaFX application has a `RadioButton` control named `radio1`, and a `Label` control named `outputLabel`. Write a lambda expression that registers an event handler with the `RadioButton` control. The event handler should display the string “selected” in `outputLabel`.
11. Write code that creates a `ListView` control named `snackListView`. Add the following strings to the `ListView`: “cheese”, “olives”, “crackers”.
12. Write code that creates a `TextArea` displaying 10 rows and 15 columns. The text area should perform line wrapping.
13. Write code that creates a `Slider` control. The control should be horizontally oriented and its range should be 0 through 1000. Labels and tick marks should be displayed. Major tick marks should appear at every 100th number, and minor tick marks should appear at every 25th number. The initial value of the `Slider` should be set at 500.

14. Write the code that creates a menu bar with one menu named File. The File menu should have the F key assigned as a mnemonic. The File menu should have three menu items: Open, Print, and Exit. Assign mnemonic keys of your choice to each of these items.

### Short Answer

1. In CSS, what is the difference between a type selector and an ID selector?
2. If you want to apply styles to all of the nodes in a scene, what selector would you use in a style definition?
3. You want the user to be able to select only one item from a group of items. Would you use `RadioButton` controls or `CheckBox` controls?
4. You want the user to be able to select any number of items from a group of items. Would you use `RadioButton` controls or `CheckBox` controls?
5. What is the purpose of a `ToggleGroup`?
6. Do you normally add `RadioButton` controls, `CheckBox` controls, or both to a `ToggleGroup`?
7. What is the difference between an editable `ComboBox` and an uneditable `ComboBox`?
8. What selection mode would you select if you want the user to only select a single item in a `ListView`?
9. You want to provide 20 items in a list for the user to select from. Which control would take up less space on the screen, a `ListView` or a `ComboBox`?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Dorm and Meal Plan Calculator

VideoNote
The Dorm and Meal Plan Calculator Problem

A university has the following dormitories:

- Allen Hall: $1,800 per semester
- Pike Hall: $2,200 per semester
- Farthing Hall: $2,800 per semester
- University Suites: $3,000 per semester

The university also offers the following meal plans:

- 7 meals per week: $600 per semester
- 14 meals per week: $1,100 per semester
- Unlimited meals: $1,800 per semester

Create an application with two `ComboBox` controls. One should hold the names of the dormitories, and the other should hold the meal plans. The user should select a dormitory and a meal plan, and the application should show the total charges for the semester.

### 2. Skateboard Designer

The Skate Shop sells the skateboard products listed in Table 13-9.

**Table 13-9** Skateboard products

| Decks | Truck Assemblies | Wheels |
|---|---|---|
| The Master Thrasher $60 | 7.75-inch axle $35 | 51 mm $20 |
| The Dictator $45 | 8-inch axle $40 | 55 mm $22 |
| The Street King $50 | 8.5-inch axle $45 | 58 mm $24 |
| | | 61 mm $28 |

In addition, the Skate Shop sells the following miscellaneous products:

- Grip tape: $10
- Bearings: $30
- Riser pads: $2
- Nuts & bolts kit: $3

Create an application that allows the user to select one deck, one truck assembly, and one wheel set from either `ListView` controls or `ComboBox` controls. The application should also have a `ListView` that allows the user to select multiple miscellaneous products. The application should display the subtotal, the amount of sales tax (at 7 percent), and the total of the order.

### 3. Conference Registration System

Create an application that calculates the registration fees for a conference. The general conference registration fee is $895 per person, and student registration is $495 per person. There is also an optional opening night dinner with a keynote speech for $30 per person. In addition, the optional preconference workshops listed in Table 13-10 are available.

**Table 13-10** Optional preconference workshops

| Workshop | Fee |
|---|---|
| Introduction to E-commerce | $295 |
| The Future of the Web | $295 |
| Advanced Java Programming | $395 |
| Network Security | $395 |

The application should allow the user to select the registration type, the optional opening night dinner and keynote speech, and as many preconference workshops as desired. The total cost should be displayed.

### 4. Smartphone Packages

Cell Solutions, a cell phone provider, sells the following data plans:

- 8 gigabytes per month: $45.00 per month
- 16 gigabytes per month: $65.00 per month
- 20 gigabytes per month: $99.00 per month

The provider sells the following phones. (A 6 percent sales tax applies to the sale of a phone.)

- Model 100: $299.95
- Model 110: $399.95
- Model 200: $499.95

Customers may also select the following options:

- Phone Replacement Insurance: $5.00 per month
- WiFi Hotspot Capability: $10.00 per month

Write an application that displays a menu system. The menu system should allow the user to select one data plan, one phone, and any of the options desired. As the user selects items from the menu, the application should show the prices of the items selected.

### 5. Shopping Cart System

Create an application that works like a shopping cart system for an online book store. In the book's source code (available at www.pearsonhighered.com/gaddis) you will find a file named *BookPrices.txt*. This file contains the names and prices of various books, formatted in the following fashion:

> *I Did It Your Way,* 11.95
> *The History of Scotland,* 14.50
> *Learn Calculus in One Day,* 29.95
> *Feel the Stress,* 18.50

Each line in the file contains the name of a book, followed by a comma, followed by the book's retail price. When your application begins execution, it should read the contents of the file and store the book titles in a `ListView`. The user should be able to select a title from the list and add it to a "shopping cart," which is simply another `ListView` control. The application should have buttons or menu items that allow the user to remove items from the shopping cart, clear the shopping cart of all selections, and check out. When the user checks out, the application should calculate and display the subtotal of all the books in the shopping cart, the sales tax (which is 7 percent of the subtotal), and the total.

CHAPTER 14

# JavaFX: Graphics, Effects, and Media

TOPICS

14.1 Drawing Shapes
14.2 Animation
14.3 Effects
14.4 Playing Sound Files
14.5 Playing Videos
14.6 Handling Key Events
14.7 Handling Mouse Events
14.8 Common Errors to Avoid

## 14.1 Drawing Shapes

**CONCEPT: JavaFX provides several classes that can be used to draw simple shapes, such as lines, circles, rectangles, and more.**

VideoNote
Drawing Shapes with JavaFX

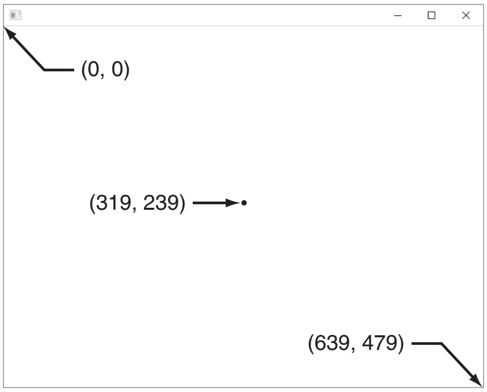
JavaFX provides several classes for drawing simple 2D shapes. In this section, we will discuss classes for drawing lines, circles, rectangles, ellipses, arcs, polygons, polylines, and text. Before we examine how to draw these shapes, however, we must discuss the screen coordinate system. You use the *screen coordinate system* to specify the location of your graphics.

### The Screen Coordinate System

The images displayed on a computer screen are made up of tiny dots called *pixels*. The screen coordinate system is used to identify the position of each pixel in an application's window. Each pixel has an *X* coordinate and a *Y* coordinate. The *X* coordinate identifies the pixel's horizontal position, and the *Y* coordinate identifies its vertical position. The coordinates are usually written in the form (*X*, *Y*). For example, the coordinates of the pixel in the upper-left corner of the screen are (0, 0). This means that its *X* coordinate is 0, and its *Y* coordinate is 0.

The *X* coordinates increase from left to right, and the *Y* coordinates increase from top to bottom. In a window that is 640 pixels wide by 480 pixels high, the coordinates of the pixel at the bottom right corner of the window are (639, 479). In the same window, the coordinates of the pixel in the center of the window are (319, 239). Figure 14-1 shows the coordinates of various pixels in the window.

**Figure 14-1** Various pixel locations in a 640 by 480 window (Oracle Corporate Counsel)



**NOTE:** The screen coordinate system differs from the Cartesian coordinate system that you learned about in mathematics. In the Cartesian coordinate system, the *Y* coordinates decrease as you move downward. In the screen coordinate system, the *Y* coordinates increase as you move downward, toward the bottom of the screen.

## The `Shape` Class and Its Subclasses

The JavaFX library has a `Shape` class that provides the basic functionality for drawing shapes. The `Shape` class has several subclasses, each of which draws a specific shape. For example, there is a `Line` class for drawing lines, a `Circle` class for drawing circles, and so forth. Table 14-1 lists some of subclasses of the `Shape` class.

**Table 14-1** Some of the Shape subclasses

| Class | Description |
|---|---|
| `Line` | Draws a line from one point to another. |
| `Circle` | Draws a circle with its center point located at a specific coordinate, and with a specified radius. |
| `Rectangle` | Draws a rectangle with a specified width and height. |
| `Ellipse` | Draws an ellipse with a specified center point, *X* radius, and *Y* radius. |
| `Arc` | Draws an arc, which is a partial ellipse. |
| `Polygon` | Draws a polygon with vertices at specified locations. |
| `Polyline` | Draws a polyline with vertices at specified locations. |
| `Text` | Draws a string at a specified location. |

The `Shape` class and its subclasses are in the `javafx.scene.shape` package. You draw shapes with these classes by following this general procedure:

1. Create an instance of the desired shape class (`Line`, `Circle`, and so forth), passing the necessary constructor arguments.
2. Repeat step 1 for each shape that you want to draw.
3. Add all of the shape objects that you created to a container.
4. Add the container to the scene graph.

Notice in step 3 you add the shape objects to a container. You can use layout containers for this purpose, but we will use `Pane` objects because they are simple containers that do not perform layout. The `Pane` class is in the `javafx.scene.layout` package.

## The `Line` Class

The `Line` class has the two constructors that are summarized in Table 14-2.

**Table 14-2** The `Line` class constructors

| Constructor | Description |
|---|---|
| `Line()` | Creates an empty line. Call the `Line` class's `setStartX` and `setStartY` methods to establish the line's starting point, and the `setEndX` and `setEndY` methods to establish the line's ending point |
| `Line(`*`startX`*`, `*`startY`*`, `*`endX`*`, `*`endY`*`)` | All of the arguments are `double`s. The *`startX`* and *`startY`* arguments are the *X* and *Y* coordinates for the line's starting point. The *endX* and *endY* arguments are the *X* and *Y* coordinates for the line's ending point. |

The second constructor is the one you will use most often. The following statement shows an example. It creates a line starting at the point (80, 120) and ending at the point (400, 520):

```
Line myLine = new Line(80, 120, 400, 520);
```

If you use the no-arg constructor, you have to use the class's `setStartX`, `setStartY`, `setEndX`, and `setEndY` methods to establish the line's starting and ending points. Here is an example that creates a line starting at the point (0, 0) and ending at the point (200, 200):

```
Line myLine = new Line();
myLine.setStartX(0);
myLine.setStartY(0);
myLine.setEndX(200);
myLine.setEndY(200);
```

Code Listing 14-1 shows a program that uses the `Line` class to draw a triangle. The programs output is shown in Figure 14-2. Figure 14-3 shows the coordinates of the triangle's corners.

**Code Listing 14-1** `(Triangle.java)`

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;

public class Triangle extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 320.0;
      final double SCENE_HEIGHT = 240.0;

      // Constants for the triangle corners.
      final double TOP_X = 160.0;
      final double TOP_Y = 20.0;
      final double BOTTOM_RIGHT_X = 310.0;
      final double BOTTOM_RIGHT_Y = 220.0;
      final double BOTTOM_LEFT_X = 20.0;
      final double BOTTOM_LEFT_Y = 220.0;

      // Draw the first line, from the top of the triangle
      // to the bottom-right corner.
      Line line1  = new Line(TOP_X, TOP_Y, BOTTOM_RIGHT_X, BOTTOM_RIGHT_Y);

      // Draw the second line, from the top of the triangle
      // to the bottom-left corner.
      Line line2  = new Line(TOP_X, TOP_Y, BOTTOM_LEFT_X, BOTTOM_LEFT_Y);

      // Draw the third line, from the bottom-left corner
      // of the triangle to the bottom-right corner.
      Line line3  = new Line(BOTTOM_LEFT_X, BOTTOM_LEFT_Y,
                             BOTTOM_RIGHT_X, BOTTOM_RIGHT_Y);

      // Add the lines to a Pane.
      Pane pane = new Pane(line1, line2, line3);

45       // Create a Scene and display it.
46       Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
47       primaryStage.setScene(scene);
48       primaryStage.show();
49    }
50 }
```

Let's take a closer look at the program:

- Lines 18 and 19 define constants for the scene's width and height.
- Lines 22 and 23 define the constants `TOP_X` and `TOP_Y`, that are used as the (*X*,*Y*) coordinates for the top corner of the triangle.
- Lines 24 and 25 define the constants `BOTTOM_RIGHT_X` and `BOTTOM_RIGHT_Y`, that are used as the (*X*,*Y*) coordinates for the bottom-right corner of the triangle.
- Lines 26 and 27 define the constants `BOTTOM_LEFT_X` and `BOTTOM_LEFT_Y`, that are used as the (*X*,*Y*) coordinates for the bottom-left corner of the triangle.
- The statement in line 31 creates a line that starts at the top corner of the triangle (160, 20) and ends at the bottom-right corner of the triangle (310, 220).
- The statement in line 35 creates a line that starts at the top corner of the triangle (160, 20) and ends at the bottom-left corner of the triangle (20, 220).
- The statement in lines 39 and 40 creates a line that starts at the bottom-left corner of the triangle (20, 220) and ends at the bottom-right corner of the triangle (310, 220).
- The statement in line 43 creates a `Pane` object and adds the three `Line` objects to it.
- The statement in line 46 creates a `Scene` object containing the `Pane`. The scene is 320 pixels wide by 240 pixels high.
- The statement in line 47 sets the scene to the stage.
- The statement in line 48 displays the application's window.

**Figure 14-2** Output of `Triangle.java` (Oracle Corporate Counsel)

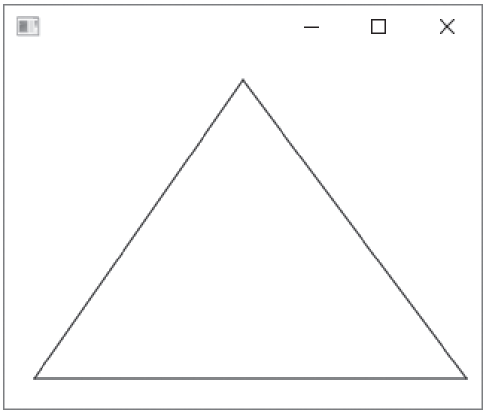
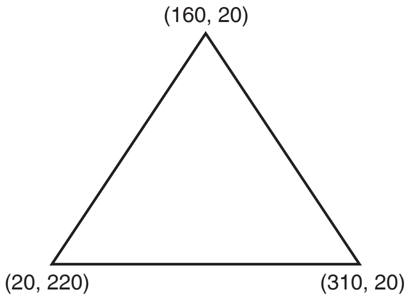


**Figure 14-3** Coordinates of the triangle's corners



## Changing the Stroke Color

The default color of lines and other shapes is black. To change a shape's color to a different value, call the `setStroke` method, which is inherited from the `Shape` class. The method's general format is

```
setStroke(color)
```

where the *`color`* argument is an object of the `Paint` class. Usually, you will pass one of the `Color` class's constants as an argument. The `Color` class, which is in the `javafx.scene.paint` package, defines several static constant fields that represent standard colors. Some examples are `Color.WHITE`, `Color.RED`, `Color.BLUE`, `Color.BLACK`, and so forth. (There are constants for other, more interesting, colors as well, such as `Color.ALICEBLUE` and `Color.BLANCHEDALMOND`. For a complete list of the fields, see the Java API documentation for the `Color` class.)

The following code snippet shows an example of creating a `Line` object and setting its stroke color to red:

```
Line myLine = new Line(80, 120, 400, 520);
myLine.setStroke(Color.RED);
```

**TIP:** Remember, to use the `Color` class, you must import it with the following statement:

```
import javafx.scene.paint.Color;
```

## The `Circle` Class

The `Circle` class has five constructors, summarized in Table 14-3.

The following statement creates a circle with its center point at (200, 200) and with a radius of 150:

```
Circle myCircle = new Circle(200, 200, 150);
```

You can use the `Circle` class's `setCenterX`, `setCenterY`, `setRadius`, and `setFill` methods to establish the circle's center point, radius, and fill color. Here is an example

**Table 14-3** The `Circle` class constructors

| Constructor | Description |
|---|---|
| `Circle()` | Creates an empty circle. Call the `Circle` class's `setCenterX` and `setCenterY` methods to establish the circle's center point, the `setRadius` method to establish the circle's radius, and the `setFill` method to fill the circle with a color. |
| `Circle(`*`radius`*`)` | The *`radius`* argument is a `double`. Creates a circle of the specified radius. Call the `Circle` class's `setCenterX` and `setCenterY` methods to establish the circle's center point, and the `setFill` method to fill the circle with a color. |
| `Circle(`*`centerX`*`, `*`centerY`*`, `*`radius`*`)` | The arguments are `double`s. Creates a circle at the specified center point, with the specified radius. By default, the circle is filled with the color black. Call the `Circle` class's `setFill` method to fill the circle with a different color. |
| `Circle(`*`centerX`*`, `*`centerY`*`, `*`radius`*`, `*`fill`*`)` | The *`centerX`*, *`centerY`*, and *`radius`* arguments are `double`s, and the *`fill`* argument is a `Paint` class object (typically a `Color` class constant). Creates a circle at the specified center point, with the specified radius, and filled with the specified color. |
| `Circle(`*`radius`*`, `*`fill`*`)` | The *`radius`* argument is a `double`, and the *`fill`* argument is a `Paint` class object (typically a `Color` class constant). Creates a circle with the specified radius, and filled with the specified color. Call the `Circle` class's `setCenterX` and `setCenterY` methods to establish the circle's center point, and the `setRadius` method to establish the circle's radius. |

that creates a circle with its center point at (75, 100), a radius of 50, and filled with the color red:

```
Circle myCircle = new Circle();
myCircle.setCenterX(75);
myCircle.setCenterY(100);
myCircle.setRadius(50);
myCircle.setFill(Color.RED);
```

If you prefer the circle to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `setStroke` method to change the color of the circle's outline. Here is an example that draws a black outline of a circle with no fill color:

```
Circle myCircle = new Circle(75, 100, 50);
myCircle.setFill(null);
myCircle.setStroke(Color.BLACK);
```

Code Listing 14-2 shows a program that uses the `Circle` class to draw a bull's eye. The bull's eye is made of four concentric circles filled with the alternating colors black and red. The program's output is shown in Figure 14-4.

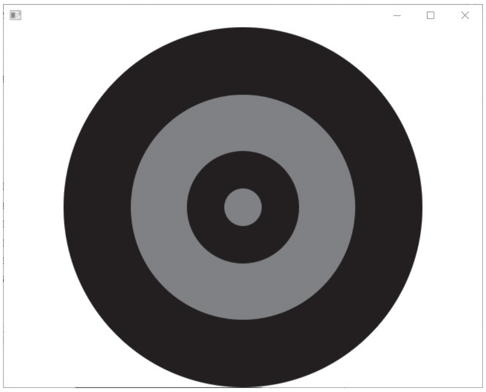
**Code Listing 14-2** (`BullsEye.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;

public class BullsEye extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 640.0;
      final double SCENE_HEIGHT = 480.0;

      // Constants for the bull's eye center point and radii.
      final double CENTER_X = 320.0;
      final double CENTER_Y = 240.0;
      final double RING1_RADIUS = 240.0;
      final double RING2_RADIUS = 150.0;
      final double RING3_RADIUS = 75.0;
      final double RING4_RADIUS = 25.0;

      // Create the circles.
      Circle ring1  = new Circle(CENTER_X, CENTER_Y,
                                 RING1_RADIUS, Color.BLACK);

      Circle ring2  = new Circle(CENTER_X, CENTER_Y,
                                 RING2_RADIUS, Color.RED);

      Circle ring3  = new Circle(CENTER_X, CENTER_Y,
                                 RING3_RADIUS, Color.BLACK);

      Circle ring4  = new Circle(CENTER_X, CENTER_Y,
                                 RING4_RADIUS, Color.RED);
42
43          // Add the circles to a Pane.
44          Pane pane = new Pane(ring1, ring2, ring3);
45
46          // Create a Scene and display it.
47          Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
48          primaryStage.setScene(scene);
49          primaryStage.show();
50      }
51  }
```

Let’s take a closer look at the program:

- Lines 19 and 20 define constants for the scene’s width and height.
- Lines 23 and 24 define the constants `CENTER_X` and `CENTER_Y`, that are used as the (*X*,*Y*) coordinates for the center of the bull’s eye.
- Lines 25 through 28 define the constants `RING1_RADIUS`, `RING2_RADIUS`, `RING3_RADIUS`, and `RING4_RADIUS`. These values are the radii of the four circles.
- Lines 31 and 32 create the first circle, filled with the color black.
- Lines 34 and 35 create the second circle, on top of the first circle, filled with the color red.
- Lines 37 and 38 create the third circle, on top of the previous circles, filled with the color black.
- Lines 40 and 41 create the fourth circle, on top of the previous circles, filled with the color red.
- The statement in line 44 creates a `Pane` object and adds the four `Circle` objects to it.
- The statement in line 47 creates a `Scene` object containing the `Pane`. The scene is 640 pixels wide by 480 pixels high.
- The statement in line 48 sets the scene to the stage.
- The statement in line 49 displays the application’s window.

**Figure 14-4** Output of `BullsEye.java` (Oracle Corporate Counsel)



## The `Rectangle` Class

The `Rectangle` class has four constructors, summarized in Table 14-4.

**Table 14-4** The `Rectangle` class constructors

| Constructor | Description |
|---|---|
| `Rectangle()` | Creates an empty rectangle. Call the `Rectangle` class's `setX` and `setY` methods to establish the position of the rectangle's upper-left corner, the `setWidth` method to establish the rectangle's width, the `setHeight` method to establish the rectangle's height, and the `setFill` method to fill the rectangle with a color. (By default, the rectangle is filled with black.) |
| `Rectangle(`*`width, height`*`)` | The arguments are `double`s. Creates a rectangle of the specified width and height. Call the `Rectangle` class's `setX` and `setY` methods to establish the position of the rectangle's upper-left corner, and the `setFill` method to fill the rectangle with a color. (By default, the rectangle is filled with black.) |
| `Rectangle(`*`X, Y, width, height`*`)` | The arguments are `double`s. Creates a rectangle with its upper-left corner at the specified (*X*,*Y*) coordinates, and of the specified width and height. Call the `Rectangle` class's `setFill` method to fill the rectangle with a color. (By default, the rectangle is filled with black.) |
| `Rectangle(`*`width, height, fill`*`)` | The *`width`* and *`height`* arguments are `double`s, and the *`fill`* argument is a `Paint` class object (typically a `Color` class constant). Creates a rectangle of the specified width and height, and filled with the specified color. Call the `Rectangle` class's `setX` and `setY` methods to establish the position of the rectangle's upper-left corner. |

The following statement creates a rectangle with its upper-left corner at (200, 100), with a width of 75 and a height of 150:

```
Rectangle myRectangle = new Rectangle(200, 100, 75, 150);
```

You can use the `Rectangle` class's `setX` and `setY` methods to establish the position of the rectangle's upper-left corner, the `setWidth` method to establish the rectangle's width, the `setHeight` method to establish the rectangle's height, and the `setFill` method to fill the rectangle with a specific color. Here is an example that creates a rectangle with its upper-left corner at (10, 20), a width of 50, a height of 100, and filled with the color dark green:

```
Rectangle myRectangle = new Rectangle();
myRectangle.setX(10);
myRectangle.setY(20);
myRectangle.setWidth(50);
myRectangle.setHeight(100);
myRectangle.setFill(Color.DARKGREEN);
```

If you prefer the rectangle to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `setStroke` method to change the color of the rectangle's outline. Here is an example that draws a black outline of a rectangle with no fill color:

```
Rectangle myRectangle = new Rectangle(200, 100, 75, 150);
myRectangle.setFill(null);
myRectangle.setStroke(Color.BLACK);
```

Code Listing 14-3 demonstrates how the `Rectangle` class can be used to create a checkerboard pattern. The program creates eight rectangles of the same size, filled with the default color black. The rectangles are displayed in a checkerboard pattern on a white background. The program's output is shown in Figure 14-5.

**Code Listing 14-3** (`CheckerBoard.java`)

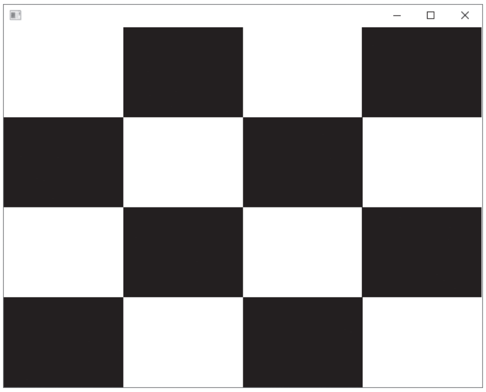
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Rectangle;

public class CheckerBoard extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 640.0;
      final double SCENE_HEIGHT = 480.0;

      // Constants for all box's width and height
      final double WIDTH = 160.0;
      final double HEIGHT = 120.0;

      // Constants for each box's (X,Y) coordinates
      final double BOX1_X = 160.0;
      final double BOX1_Y = 0.0;

      final double BOX2_X = 479.0;
      final double BOX2_Y = 0.0;

      final double BOX3_X = 0.0;
      final double BOX3_Y = 120.0;

35       final double BOX4_X = 320.0;
36       final double BOX4_Y = 120.0;
37
38       final double BOX5_X = 160.0;
39       final double BOX5_Y = 240.0;
40
41       final double BOX6_X = 479.0;
42       final double BOX6_Y = 240.0;
43
44       final double BOX7_X = 0.0;
45       final double BOX7_Y = 360.0;
46
47       final double BOX8_X = 320.0;
48       final double BOX8_Y = 360.0;
49
50       // Create the boxes.
51       Rectangle box1  = new Rectangle(BOX1_X, BOX1_Y, WIDTH, HEIGHT);
52       Rectangle box2  = new Rectangle(BOX2_X, BOX2_Y, WIDTH, HEIGHT);
53       Rectangle box3  = new Rectangle(BOX3_X, BOX3_Y, WIDTH, HEIGHT);
54       Rectangle box4  = new Rectangle(BOX4_X, BOX4_Y, WIDTH, HEIGHT);
55       Rectangle box5  = new Rectangle(BOX5_X, BOX5_Y, WIDTH, HEIGHT);
56       Rectangle box6  = new Rectangle(BOX6_X, BOX6_Y, WIDTH, HEIGHT);
57       Rectangle box7  = new Rectangle(BOX7_X, BOX7_Y, WIDTH, HEIGHT);
58       Rectangle box8  = new Rectangle(BOX8_X, BOX8_Y, WIDTH, HEIGHT);
59
60       // Add the boxes to a Pane.
61       Pane pane = new Pane(box1, box2, box3, box4,
62                            box5, box6, box7, box8);
63
64       // Create a Scene and display it.
65       Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
66       primaryStage.setScene(scene);
67       primaryStage.show();
68    }
69 }
```

Let's take a closer look at the program:

- Lines 18 and 19 define constants for the scene's width and height.
- Lines 22 and 23 define the constants `WIDTH` and `HEIGHT`, which are used as the width and height for each box that makes up the checkerboard.
- Lines 26 through 48 define constants for the ($X$,$Y$) coordinates for the upper-left corner of each rectangle.
- Lines 51 through 58 create the eight `Rectangle` objects. By default, they are filled with the color black.
- The statement in lines 61 and 62 creates a `Pane` object and adds the eight `Rectangle` objects to it.

- The statement in line 65 creates a `Scene` object containing the `Pane`. The scene is 640 pixels wide by 480 pixels high.
- The statement in line 66 sets the scene to the stage.
- The statement in line 67 displays the application's window.

**Figure 14-5** Output of CheckerBoard.java (Oracle Corporate Counsel)



## The `Ellipse` Class

The `Ellipse` class draws an ellipse, which is an oval shape. The `Ellipse` class has three constructors, summarized in Table 14-5.

**Table 14-5** The `Ellipse` class constructors

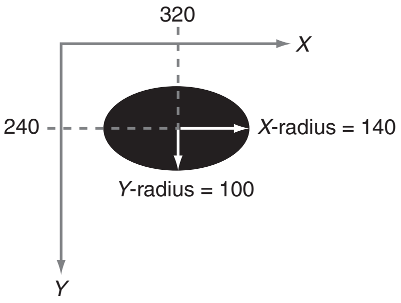
| Constructor | Description |
|---|---|
| `Ellipse()` | Creates an empty ellipse. Call the `Ellispe` class's `setCenterX` and `setCenterY` methods to establish the ellipse's center point, the `setRadiusX` method to establish the ellipse's radius along the *X* axis, and the `setRadiusY` method to establish the ellipse's radius along the *Y* axis. |
| `Ellipse(`*`radiusX`*`, `*`radiusY`*`)` | The arguments are `double`s. Creates an ellipse with the specified radii along the *X* and *Y* axes. Call the `Ellispe` class's `setCenterX` and `setCenterY` methods to establish the ellipse's center point. |
| `Ellipse(`*`centerX`*`, `*`centerY`*`, `*`radiusX`*`, `*`radiusY`*`)` | The arguments are `double`s. Creates an ellipse at the specified center point, with the specified *X* and *Y* radii. |

Here is an example:

```
Ellipse myEllipse = new Ellipse(320, 240, 140, 100);
```

As shown in Figure 14-6, the ellipse created by this statement will have its center located at (320, 240), an *X*-radius of 140 pixels, and a *Y*-radius of 100. The use of two different radii, one for the *X* axis and one for the *Y* axis, gives the ellipse its shape.

**Figure 14-6** An ellipse's center point, x-radius, and y-radius



You can use the `Ellipse` class's `setCenterX`, `setCenterY`, `setRadiusX`, and `setRadiusY` methods to establish the ellipse's center point, *X*-radius, and *Y*-radius. By default, the ellipse will be filled with the color black. You can call the `Ellipse` class's `setFill` method (inherited from the `Shape` class) to fill it with another color. If you prefer the ellipse to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `Ellipse` class's `setStroke` method (also inherited from the `Shape` class) to change the color of the ellipse's outline.

The following code shows an example. It creates an ellipse with its center point at (125, 100), an *X*-radius of 130, a *Y*-radius of 90, no fill color, and a stroke color of black:

```
Ellipse myEllipse = new Ellipse();
myEllipse.setCenterX(125);
myEllipse.setCenterY(100);
myEllipse.setRadiusX(130);
myEllipse.setRadiusY(90);
myEllipse.setFill(null);
myEllipse.setStroke(Color.BLACK);
```

The program shown in Code Listing 14-4 draws a blue circle representing a planet, and a gray circle representing a moon. An ellipse is also drawn representing the moon's orbital path around the planet. The program's output is shown in Figure 14-7.

**Code Listing 14-4** `(Orbit.java)`

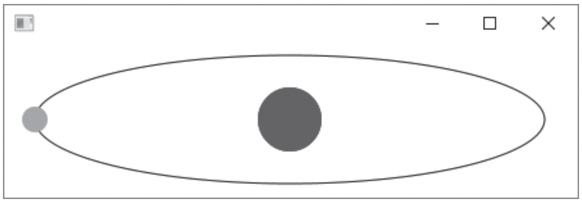
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Ellipse;

```

```
public class Orbit extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 450.0;
      final double SCENE_HEIGHT = 120.0;

      // Constants for the planet, orbit, and moon
      final double PLANET_X = 224.0;
      final double PLANET_Y = 59.0;
      final double PLANET_RAD = 25.0;

      final double ORBIT_RAD_X = 200.0;
      final double ORBIT_RAD_Y = 50.0;

      final double MOON_X = 24.0;
      final double MOON_Y = 59.0;
      final double MOON_RAD = 10.0;

      // Draw the planet.
      Circle planet = new Circle(PLANET_X, PLANET_Y,
                                 PLANET_RAD, Color.BLUE);

      // Draw the elliptical orbit path.
      Ellipse orbitPath = new Ellipse(PLANET_X, PLANET_Y,
                                      ORBIT_RAD_X, ORBIT_RAD_Y);
      orbitPath.setFill(null);
      orbitPath.setStroke(Color.BLACK);

      // Draw the moon.
      Circle moon = new Circle(MOON_X, MOON_Y,
                               MOON_RAD, Color.DARKGRAY);

      // Add the boxes to a Pane.
      Pane pane = new Pane(planet, orbitPath, moon);

      // Create a Scene and display it.
      Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

Let's take a closer look at the program:

- Lines 20 and 21 define constants for the scene's width and height.
- Lines 24 through 26 define constants for the planet's center point and radius.
- Lines 28 and 29 define constants for the orbit's *X*-radius and *Y*-radius.
- Lines 31 through 33 define constants for the moon's center point and radius.
- Lines 36 and 37 create the planet as a circle filled with the color blue.
- Lines 40 and 41 create the orbital path as an ellipse. The center point of the ellipse is the same as the center point of the planet.
- Line 42 sets the ellipse's fill color to `null`. As a result, the ellipse will not be filled with a color.
- Line 43 sets the ellipse's stroke color to black.
- Lines 46 and 47 create the moon as a circle filled with the color dark gray.
- Line 50 creates a `Pane`, and adds the `planet`, `orbitPath`, and `moon` objects to it.
- Line 53 creates a `Scene` object containing the `Pane`. The scene is 450 pixels wide by 120 pixels high.
- Line 54 sets the scene to the stage.
- Line 55 displays the application's window.

**Figure 14-7** Output of `Orbit.java` (Oracle Corporate Counsel)



**NOTE:** If an ellipse's *X*-radius and *Y*-radius have the same value, then the ellipse will be a circle.

## The `Arc` Class

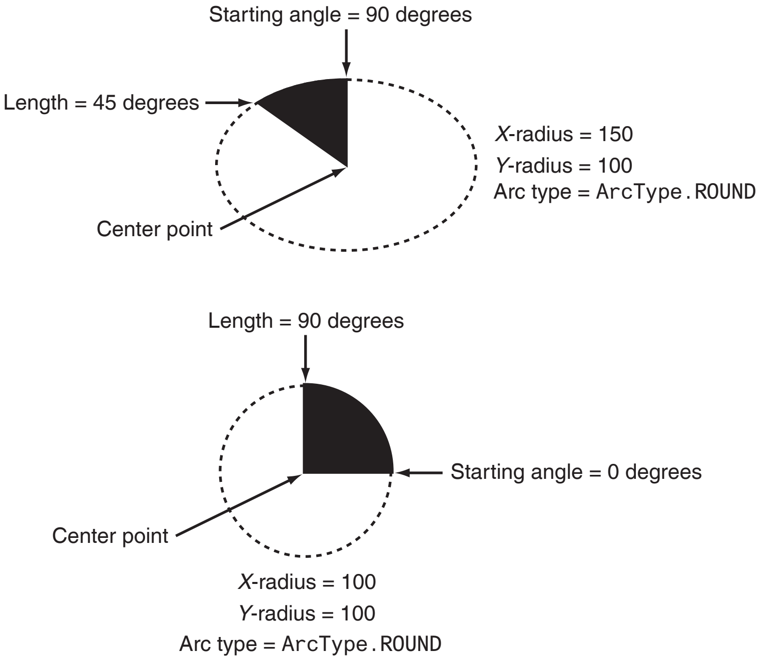
The `Arc` class draws an arc, which is considered to be part of an oval. The `Arc` class has two constructors, summarized in Table 14-6.

**Table 14-6** The `Arc` class constructors

| Constructor | Description |
|---|---|
| `Arc()` | Creates an empty arc. Call the `Arc` class's `setCenterX` and `setCenterY` methods to establish the arc's center point, the `setRadiusX` method to establish the arc's radius along the *X* axis, the `setRadiusY` method to establish the arc's radius along the *Y* axis, the `setStartAngle` method to establish the arc's starting angle (in degrees), and the `setLength` method to establish the arc's angular extent (in degrees). |
| `Arc(`*`centerX`*`, `*`centerY`*`, `*`radiusX`*`, `*`radiusY`*`, `*`startAngle`*`, `*`length`*`)` | The arguments are `double`s. Creates an arc at the specified center point, with the specified *X* and *Y* radii. The arc begins at the angle specified by *`startAngle`*, and extends counterclockwise. The *`length`* argument specifies the number of degrees that the arc extends from its starting angle. |

Because an arc is part of an ellipse, you need to establish a center point, an *X*-radius, and a *Y*-radius. In addition, an arc has a starting angle and an angular length, both of which are specified in degrees. Note 0 degrees is at the 3 o'clock position, and arcs are drawn counterclockwise from their starting angle. Figure 14-8 summarizes the properties of an arc with two examples.

**Figure 14-8** Arc properties


Starting angle = 90 degrees
Length = 45 degrees
X-radius = 150
Y-radius = 100
Arc type = ArcType.ROUND
Center point
Length = 90 degrees
Starting angle = 0 degrees
Center point
X-radius = 100
Y-radius = 100
Arc type = ArcType.ROUND

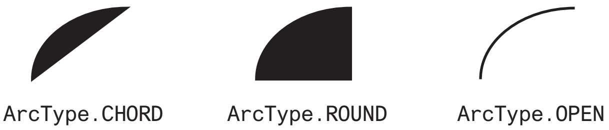
By default, the arc will be filled with the color black. You can call the `Arc` class's `setFill` method (inherited from the `Shape` class) to fill it with another color. If you prefer the arc to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `Arc` class's `setStroke` method (also inherited from the `Shape` class) to change the color of the ellipse's outline.

There are three types of arc that you can draw with the `Arc` class, summarized in Table 14-7. To select one of these arc types, you must import the `ArcType` class from the `javafx.scene.shape` package. You then pass one of the constants listed in Table 14-7 to the `Arc` class's `setType` method. (Note the default type is `ArcType.CHORD`.) Figure 14-9 shows examples of each type of arc.

**Table 14-7** Arc types

| Type | Description |
|---|---|
| `ArcType.CHORD` | This is the default arc type. A straight line will be drawn from one endpoint of the arc to the other endpoint. |
| `ArcType.ROUND` | Straight lines will be drawn from each endpoint to the arc's center point. As a result, the arc will be shaped like a pie slice. |
| `ArcType.OPEN` | No lines will connect the endpoints. Only the arc will be drawn. |

**Figure 14-9** Types of arcs



Code Listing 14-5 shows an example program that uses arcs to draw a pie chart. The program's output is shown in Figure 14-10.

**Code Listing 14-5** (PieChart.java)

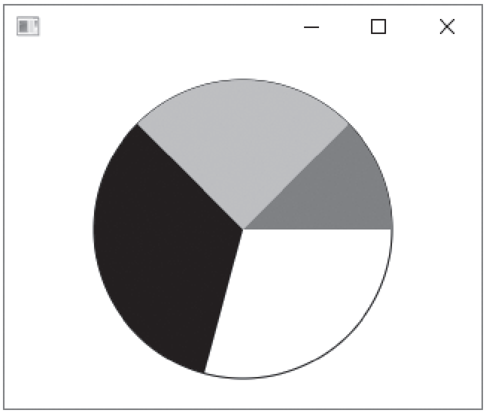
```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;

public class Arcs extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 320.0;
      final double SCENE_HEIGHT = 240.0;

      // Common attributes for the arcs
      final double X = 160.0;       // Center point X coordinate
      final double Y = 120.0;       // Center point Y coordinate
      final double X_RAD = 100.00;  // X-radius
      final double Y_RAD = 100.0;   // Y-radius

30        // Constants for the arc angles
31        final double PIE1_START = 0.0;
32        final double PIE1_LENGTH = 45.0;
33        final double PIE2_START = 45.0;
34        final double PIE2_LENGTH = 90.0;
35        final double PIE3_START = 135.0;
36        final double PIE3_LENGTH = 120.0;
37
38        // Draw a circle outline.
39        Circle outline = new Circle(X, Y, X_RAD, Color.WHITE);
40        outline.setStroke(Color.BLACK);
41
42        // Draw the first pie-slice.
43        Arc pieSlice1  = new Arc(X,Y, X_RAD, Y_RAD,
44                                 PIE1_START, PIE1_LENGTH);
45        pieSlice1.setFill(Color.RED);
46        pieSlice1.setType(ArcType.ROUND);
47
48        // Draw the second pie-slice.
49        Arc pieSlice2  = new Arc(X,Y, X_RAD, Y_RAD,
50                                 PIE2_START, PIE2_LENGTH);
51        pieSlice2.setFill(Color.LIGHTGREEN);
52        pieSlice2.setType(ArcType.ROUND);
53
54        // Draw the third pie-slice.
55        Arc pieSlice3  = new Arc(X,Y, X_RAD, Y_RAD,
56                                 PIE3_START, PIE3_LENGTH);
57        pieSlice3.setFill(Color.BLACK);
58        pieSlice3.setType(ArcType.ROUND);
59
60        // Add the objects to a Pane.
61        Pane pane = new Pane(outline, pieSlice1, pieSlice2,
62                             pieSlice3);
63
64        // Create a Scene and display it.
65        Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
66        primaryStage.setScene(scene);
67        primaryStage.show();
68     }
69  }
```

Let's take a closer look at the program:

- Lines 21 and 22 define constants for the scene's width and height.
- Lines 25 through 28 define constants for the common attributes of the arcs (center point coordinates, *X*-radius, and *Y*-radius).
- Lines 31 through 36 define constants for each of the pie slice's starting angle and angular length.

- Lines 39 and 40 create a white circle with a black outline. This will serve as the outline for the "pie."
- Lines 43 through 46 create the first pie slice, sets its fill color to red, and its type to `ArcType.ROUND`.
- Lines 49 through 52 create the second pie slice, sets its fill color to light green, and its type to `ArcType.ROUND`.
- Lines 55 through 58 create the third pie slice, sets its fill color to black, and its type to `ArcType.ROUND`.
- Lines 61 and 62 create a `Pane`, and adds the `outline`, `pieSlice1`, `pieSlice2`, and `pieSlice3` objects to it.
- Line 65 creates a `Scene` object containing the `Pane`. The scene is 320 pixels wide by 240 pixels high.
- Line 66 sets the scene to the stage.
- Line 67 displays the application's window.

**Figure 14-10** Output of `PieChart.java` (Oracle Corporate Counsel)



## The `Polygon` Class

The `Polygon` class creates a closed polygon that, by default, is filled with the color black. A polygon is constructed of multiple line segments that are connected. The point where two line segments are connected is called a *vertex*. The `Polygon` class has two constructors, summarized in Table 14-8.

**Table 14-8** The `Polygon` class constructors

| Constructor | Description |
|---|---|
| `Polygon()` | Creates an empty polygon. Call the `Polygon` class's `getPoints().addAll()` method to establish the *XY* coordinates of the polygon's vertices. |
| `Polygon(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`, ...)` | The variable number of arguments, which are `double`s, are the *XY* coordinates of the polygon's vertices. |

When you call the parameterized constructor, you pass the (*X*,*Y*) coordinates of each vertex as arguments. The first two arguments are the (*X*,*Y*) coordinates of the first vertex, the next two arguments are the (*X*,*Y*) coordinates of the second vertex, and so forth. Here is an example that draws a diamond shape:

```
Polygon diamond = new Polygon(160.0, 20.0,  // Top
                              300.0, 120.0, // Right
                              160.0, 220.0, // Bottom
                              20.0,  120.0, // Left
                              160.0, 20.0); // Top
```

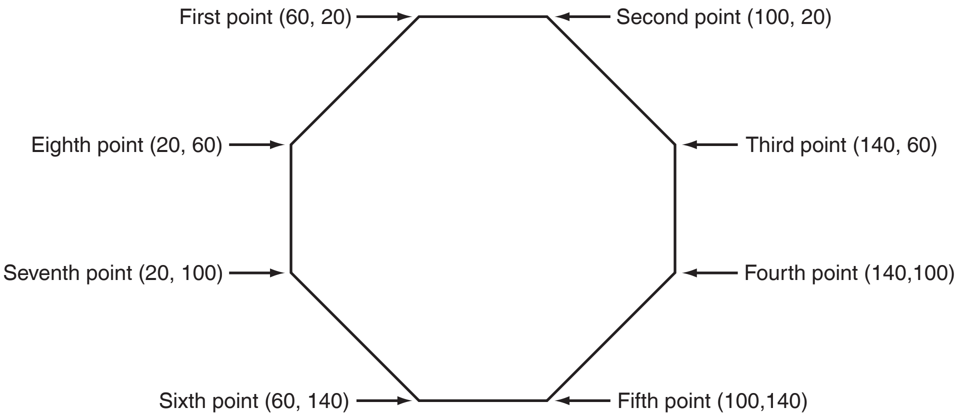
If you call the no-arg constructor to create an empty polygon, you can then call the `Polygon` class's `getPoints().addAll()` method to establish the coordinates of the vertices. Here is an example that creates the diamond shape previously shown:

```
Polygon diamond = new Polygon();
diamond.getPoints().addAll(160.0, 20.0,  // Top
                           300.0, 120.0, // Right
                           160.0, 220.0, // Bottom
                           20.0,  120.0, // Left
                           160.0, 20.0); // Top
```

By default, polygons are filled with the color black. You can call the `Polygon` class's `setFill` method (inherited from the `Shape` class) to fill it with another color. If you prefer the polygon to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `Polygon` class's `setStroke` method (also inherited from the `Shape` class) to change the color of the polygon outline.

Code Listing 14-6 shows an example program that uses the `Polygon` class to draw a red octagon with the vertices shown in Figure 14-11. The program's output is shown in Figure 14-12.

**Figure 14-11** Vertices of the octagon displayed by `Octagon.java`



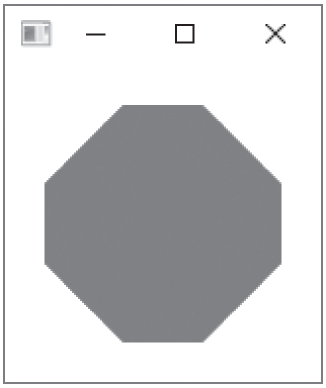
**Code Listing 14-6** (`Octagon.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;

public class Octagon extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 160.0;
      final double SCENE_HEIGHT = 160.0;

      // Constants for the vertices.
      final double X1 = 60.0,  Y1 = 20.0;
      final double X2 = 100.0, Y2 = 20.0;
      final double X3 = 140.0, Y3 = 60.0;
      final double X4 = 140.0, Y4 = 100.0;
      final double X5 = 100.0, Y5 = 140.0;
      final double X6 = 60.0,  Y6 = 140.0;
      final double X7 = 20.0,  Y7 = 100.0;
      final double X8 = 20.0,  Y8 = 60.0;

      // Create an octagon.
      Polygon octagon = new Polygon(X1, Y1, X2, Y2,
                                    X3, Y3, X4, Y4,
                                    X5, Y5, X6, Y6,
                                    X7, Y7, X8, Y8);

      // Set the octagon's fill color to red.
      octagon.setFill(Color.RED);

      // Add the octagon to a Pane.
      Pane pane = new Pane(octagon);

44          // Create a Scene and display it.
45          Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
46          primaryStage.setScene(scene);
47          primaryStage.show();
48      }
49  }
```

Let's take a closer look at the program:

- Lines 19 and 20 define constants for the scene's width and height.
- Lines 23 through 30 define constants for the *X* and *Y* coordinates of each vertex.
- Lines 33 through 36 create the octagon.
- Line 39 sets the octagon's fill color to `Color.RED`.
- Line 42 creates a `Pane`, and adds the `octagon` object to it.
- Line 45 creates a `Scene` object containing the `Pane`. The scene is 160 pixels wide by 160 pixels high.
- Line 46 sets the scene to the stage.
- Line 47 displays the application's window.

**Figure 14-12** Output of `Octogon.java` (Oracle Corporate Counsel)



## The `Polyline` Class

The `Polyline` class creates a line with multiple segments. A polyline is very similar to a polygon, except that a polyline is not automatically closed. The `Polyline` class has two constructors, summarized in Table 14-9.

**Table 14-9** The `PolyLine` class constructors

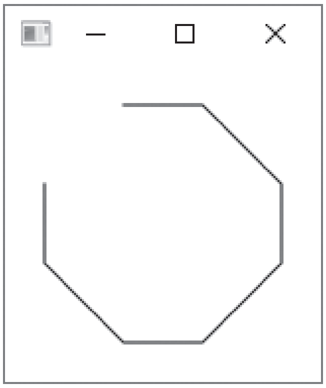
| Constructor | Description |
|---|---|
| `Polyline()` | Creates an empty polyline. Call the `Polyline` class's `getPoints().addAll()` method to establish the *XY* coordinates of the polyline's vertices. |
| `Polyline(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`, . . .)` | The variable number of arguments, which are `double`s, are the *XY* coordinates of the polyline's vertices. |

To demonstrate the similarities between the `Polyline` class and the `Polygon` class, look at Code Listing 14-7. It creates a polyline using the same vertex coordinates as the *Octagon.java* program in Code Listing 14-6. The program's output is shown in Figure 14-13.

### Code Listing 14-7 (PolylineDemo.java)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Polyline;

public class PolylineDemo extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 160.0;
      final double SCENE_HEIGHT = 160.0;

      // Constants for the vertices.
      final double X1 = 60.0,  Y1 = 20.0;
      final double X2 = 100.0, Y2 = 20.0;
      final double X3 = 140.0, Y3 = 60.0;
      final double X4 = 140.0, Y4 = 100.0;
      final double X5 = 100.0, Y5 = 140.0;
      final double X6 = 60.0,  Y6 = 140.0;
      final double X7 = 20.0,  Y7 = 100.0;
      final double X8 = 20.0,  Y8 = 60.0;

      // Create a polyline.
      Polyline polyline = new Polyline(X1, Y1, X2, Y2,
                                       X3, Y3, X4, Y4,
                                       X5, Y5, X6, Y6,
                                       X7, Y7, X8, Y8);

      // Add the polyline to a Pane.
      Pane pane = new Pane(polyline);

      // Create a Scene and display it.
      Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 14-13** Output of `PolylineDemo.java` (Oracle Corporate Counsel)



## The `Text` Class

The `Text` class draws a string on the screen as a graphic. Because the `Text` class is a subclass of the `Shape` class, it allows you to change its fill color, stroke, and other characteristics, just like any of the other shapes. The `Text` class has three constructors, summarized in Table 14-10.

**Table 14-10** The `Text` class constructors

| Constructor | Description |
|---|---|
| `Text()` | Creates an empty `Text` object. Call the `Text` class's `setX` and `setY` methods to establish the `Text` object's location, and the `setText` method to establish the string that the object should display. |
| `Text(`*`x`*`, `*`y`*`, `*`text`*`)` | The *`x`* and *`y`* arguments, which are `double`s, are the *XY* coordinates of the object's bottom-left corner. The *`text`* argument is the string that the object will display. |
| `Text(`*`text`*`)` | The *`text`* argument is the string that the object will display. Call the `Text` class's `setX` and `setY` methods to establish the `Text` object's location. |

The following statement draws the string "Hello World", starting at the coordinates 100, 50:

```
Text myText = new Text(100.0, 50.0, "Hello World");
```

You can set the font for the string with the `setFont` method. This method accepts a `Font` object as its argument. The `Font` class is in the `javafx.scene.text` package. When you instantiate the `Font` class, you pass the name of a font and the font's size, in points, as arguments to the constructor. Here is an example that creates a `Text` object, sets the font to Serif 36 points, sets the stroke color to black, and sets the fill color to red:

```
myText.setFont(new Font("Serif", 36));
myText.setStroke(Color.BLACK);
myText.setFill(Color.RED);
```

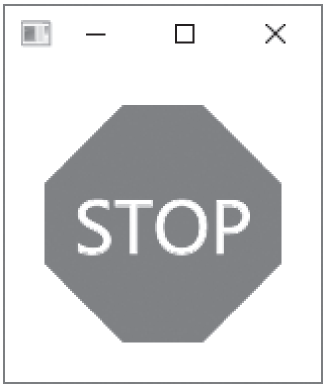
Code Listing 14-8 demonstrates the Text class. It draws the same octagon shape that the program in Code Listing 14-5 class displayed, then draws the string “STOP” over it to create a stop sign. The string is drawn in a bold 38-point Sans Serif font. The program’s output is shown in Figure 14-14.

**Code Listing 14-8** (`StopSign.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

public class StopSign extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 160.0;
      final double SCENE_HEIGHT = 160.0;

      // Constants for the vertices.
      final double X1 = 60.0,  Y1 = 20.0;
      final double X2 = 100.0, Y2 = 20.0;
      final double X3 = 140.0, Y3 = 60.0;
      final double X4 = 140.0, Y4 = 100.0;
      final double X5 = 100.0, Y5 = 140.0;
      final double X6 = 60.0,  Y6 = 140.0;
      final double X7 = 20.0,  Y7 = 100.0;
      final double X8 = 20.0,  Y8 = 60.0;

```

```
        // Constants for the text
        final double TEXT_X = 35.0;
        final double TEXT_Y = 95.0;
        final double FONT_SIZE = 38.0;

        // Create an octagon.
        Polygon octagon = new Polygon(X1, Y1, X2, Y2,
                                      X3, Y3, X4, Y4,
                                      X5, Y5, X6, Y6,
                                      X7, Y7, X8, Y8);

        // Set the octagon's fill color to red.
        octagon.setFill(Color.RED);

        // Create a Text object.
        Text stopText = new Text(TEXT_X, TEXT_Y, "STOP");
        stopText.setStroke(Color.WHITE);
        stopText.setFill(Color.WHITE);
        stopText.setFont(new Font("SansSerif", FONT_SIZE));

        // Add the octagon to a Pane.
        Pane pane = new Pane(octagon, stopText);

        // Create a Scene and display it.
        Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

**Figure 14-14** Output of `StopSign.java` (Oracle Corporate Counsel)



## Rotating Nodes

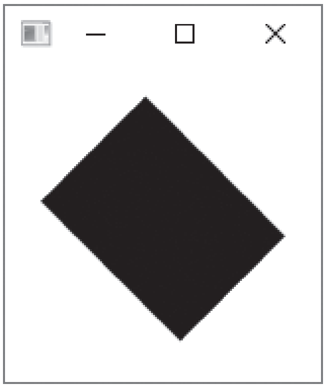
The `Node` class provides a method named `setRotate` that rotates a node about its center. Because the `setRotate` method is in the `Node` class, it can be used to rotate any node in your scene graph (including controls, such as Buttons and Labels).

When you call the method, you pass a `double` that is the angle of rotation, in degrees. The following code snippet is taken from the program *RotateBox.java* in the Student Sample Programs:

```
// Constants for the rectangle
final double X = 30.0, Y = 40.0;
final double WIDTH = 100.00, HEIGHT = 75.0;
final double ANGLE = 45.0;

// Create a rectangle.
Rectangle box = new Rectangle(X, Y, WIDTH, HEIGHT);
box.setRotate(ANGLE);
```

This code creates a rectangle, rotated at 45 degrees. The program's output is shown in Figure 14-15.

**Figure 14-15** Output of `RotateBox.java` (Oracle Corporate Counsel)



If you rotate a container, the rotation will be applied to all children in the container. For example, Code Listing 14-9 displays a rotated stop sign. This program is a modification of *StopSign.java*, that you saw in Code Listing 14-7. In this version of the program, in line 57, the root node in the scene graph (a `Pane` object) is rotated. As a result, all of the objects in the scene graph are rotated 30 degrees. The program's output is shown in Figure 14-16.

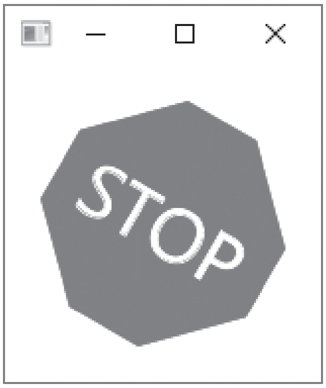
**Code Listing 14-9** (`RotateStopSign.java`)

```
1  import javafx.application.Application;
2  import javafx.stage.Stage;
3  import javafx.scene.Scene;
4  import javafx.scene.layout.Pane;
5  import javafx.scene.paint.Color;
```

```
import javafx.scene.shape.Polygon;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

public class RotateStopSign extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene
      final double SCENE_WIDTH = 160.0;
      final double SCENE_HEIGHT = 160.0;
      final double ANGLE = 30.0;

      // Constants for the vertices
      final double X1 = 60.0,  Y1 = 20.0;
      final double X2 = 100.0, Y2 = 20.0;
      final double X3 = 140.0, Y3 = 60.0;
      final double X4 = 140.0, Y4 = 100.0;
      final double X5 = 100.0, Y5 = 140.0;
      final double X6 = 60.0,  Y6 = 140.0;
      final double X7 = 20.0,  Y7 = 100.0;
      final double X8 = 20.0,  Y8 = 60.0;

      // Constants for the text
      final double TEXT_X = 35.0;
      final double TEXT_Y = 95.0;
      final double FONT_SIZE = 38.0;

      // Create an octagon.
      Polygon octagon = new Polygon(X1, Y1, X2, Y2,
                                    X3, Y3, X4, Y4,
                                    X5, Y5, X6, Y6,
                                    X7, Y7, X8, Y8);

      // Set the octagon's fill color to red.
      octagon.setFill(Color.RED);

      // Create a Text object.
      Text stopText = new Text(TEXT_X, TEXT_Y, "STOP");
      stopText.setStroke(Color.WHITE);
      stopText.setFill(Color.WHITE);
      stopText.setFont(new Font("SansSerif", FONT_SIZE));

```

```
        // Add the octagon to a Pane.
        Pane pane = new Pane(octagon, stopText);
        pane.setRotate(ANGLE);

        // Create a Scene and display it.
        Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

**Figure 14-16** Output of RotateStopSign.java (Oracle Corporate Counsel)



## Scaling Nodes

The `Node` class also provides methods named `setScaleX` and `setScaleY` that scale a node in its *X* and *Y* dimensions. Because these methods are in the `Node` class, they can be used to scale any node in your scene graph.

When you call the `setScaleX` and `setScaleY` methods, you pass a `double` that will be used as the scaling factor. For example, 2.0 will make the node twice its original size, and 0.5 will make the node half its original size. If you want the node to retain its original proportions, be sure to scale it the same in both the *X* and *Y* dimensions (in other words, pass the same value to the `setScaleX` and `setScaleY` methods). If you pass different values to these methods, the node will appear stretched in either the *X* or *Y* dimensions.

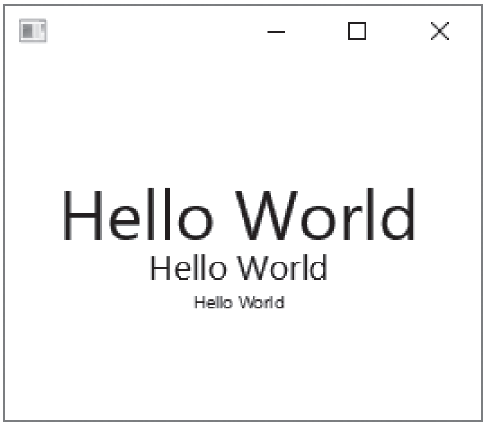
The following code snippet is taken from the program *ScaleText.java* in the Student Sample Programs:

```
// Constants for the text
final double X1  = 30.0, Y1  = 100.0;
final double X2  = 30.0, Y2  = 130.0;
final double X3  = 30.0, Y3  = 150.0;
final double FONT_SIZE = 38;
final double SCALE_HALF = 0.5;
final double SCALE_QTR = 0.25;
```

```
// Create three Text objects.
Text text1 = new Text(X1, Y1, "Hello World");
text1.setFont(new Font("SansSerif", FONT_SIZE));

Text text2 = new Text(X2, Y2, "Hello World");
text2.setFont(new Font("SansSerif", FONT_SIZE));
text2.setScaleX(SCALE_HALF);
text2.setScaleY(SCALE_HALF);

Text text3 = new Text(X3, Y3, "Hello World");
text3.setFont(new Font("SansSerif", FONT_SIZE));
text3.setScaleX(SCALE_QTR);
text3.setScaleY(SCALE_QTR);
```

This code creates three `Text` objects, each displaying the string “Hello World”. The first `Text` object is displayed at its original size. The second is scaled to half its original size, and the third is scaled to one-fourth its original size. The program’s output is shown in Figure 14-17.

**Figure 14-17** Output of `ScaleText.java` (Oracle Corporate Counsel)



```
Hello World
Hello World
Hello World
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.1 What are the coordinates of the pixel in the upper-left corner of the window?

14.2 In a window 640 pixels wide by 480 pixels high, what are the coordinates of the pixel in the lower-right corner?

14.3 How is the screen coordinate system different from the Cartesian coordinate system?

14.4 In what package is the `Shape` class?

14.5 In what package is the `Color` class?

14.6 In what package is the `Pane` class?

14.7 What shape method do you call to change the color of a shape’s outline?

14.8 What shape method do you call to change a shape’s fill color?

14.9 What shape method do you call to rotate a shape around its center?

14.10 What shape method do you call to scale a shape along the *X* axis?

14.11 What shape method do you call to scale a shape along the *Y* axis?

14.12 Which of the following is not a subclass of the `Shape` class?

a. `Circle`
b. `Rectangle`
c. `Triangle`
d. `Ellipse`

## 14.2 Animation

**CONCEPT: JavaFX provides transition classes that create animations by causing a node to change, over time, from one state to another.**

VideoNote
JavaFX Animation

JavaFX provides a number of classes, known as *transition classes*, that make it easy to create animations. The transition classes are in the `javafx.animation` package, and some of them are listed in Table 14-11.

**Table 14-11** Some of the transition classes

| Class | Description |
|---|---|
| `TranslateTransition` | Creates an animation in which a node moves from one position to another over a period of time. |
| `RotateTransition` | Creates an animation in which a node rotates over a period of time. |
| `ScaleTransition` | Creates an animation in which a node becomes larger or smaller over a period of time. |
| `StrokeTransition` | Creates an animation in which a node's stroke color changes over a period of time. |
| `FillTransition` | Creates an animation in which a node's fill color changes over a period of time. |
| `FadeTransition` | Creates an animation in which a node fades in or out over a period of time. |

Although each of the transition classes creates a different type of animation, all of the classes work in a similar manner. For each class, you designate the following:

- The node that you want to animate
- The amount of time that the animation will last
- The node's starting state (depends on the type of animation)
- The node's ending state (depends on the type of animation)

Once you have everything properly setup, you call the transition class's `play` method to play the animation. Let's look at how this relates to each of the transition classes listed in Table 14-10.

## The `TranslateTransition` Class

The `TranslateTransition` class causes a node to move from one position on the screen to another. The `TranslateTransition` class has three constructors, summarized in Table 14-12.

**Table 14-12** `TranslateTransition` class constructors

| Constructor | Description |
|---|---|
| `TranslateTransition()` | Creates an empty `TranslateTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setNode` method to specify the node to animate. |
| `TranslateTransition(`*`duration`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `TranslateTransition` class's `setNode` method to specify the node to animate. |
| `TranslateTransition(`*`duration, node`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`node`* argument, an object of the `Node` class (or its subclasses) is the node to animate. |

You use a `Duration` object to specify the amount of time that an animation should last. The `Duration` class, which is in the `javafx.util` package, represents a duration of time. When you instantiate the `Duration` class, you pass a number of milliseconds as an argument to the constructor. A millisecond is 1/1000th of a second, so passing the value 1000 to the `Duration` class constructor will create an object that represents a duration of 1 second.

The following code snippet shows an example of creating a `Circle`, with its center point at (0, 50) and a radius of 30. Then, a `TranslateTransition` object is created to animate it. The duration of the animation is 3 seconds.

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
         new TranslateTransition(new Duration(3000), myCircle);
```

The next step is to specify the node's starting location. The `TranslateTransition` class has two methods named `setFromX` and `setFromY` for this purpose. You call these methods, passing the starting location's *X* and *Y* coordinates, respectively.

Next, you specify the node's ending location by calling the `TranslateTransition` class's `setToX` and `setToY` methods. You pass the ending location's *X* and *Y* coordinates, respectively, to these methods. Once you have everything setup, you call the transition class's `play` method to play the animation. The following code snippet shows an example:

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
         new TranslateTransition(new Duration(3000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(50);
ttrans.setToX(100);
ttrans.setToY(50);
ttrans.play();
```

This code creates a `Circle`, then creates a `TranslateTransition` object to animate it. The duration of the animation is 3 seconds. When the animation is played, the `Circle` object will start at (0, 50), and it will move to the location (100, 50). It will take 3 seconds for the `Circle` object to complete the move.

Keep in mind that the values you pass to the `setToX` and `setToY` methods are the coordinates for the node's ending position. In the previous code snippet, the circle's final position is (100, 50). If you prefer, you can call the `setByX` and `setByY` methods to specify the distance that the node should move, relative to its starting position. The following code snippet shows an example:

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
         new TranslateTransition(new Duration(3000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(50);
ttrans.setByX(75);
ttrans.setByY(50);
ttrans.play();
```

In this example, the transition object initially positions the circle at (0, 50), then moves it by an additional 75 pixels in the *X* direction, and an additional 50 pixels in the *Y* direction. So, the circle's ending position will be (75, 100).

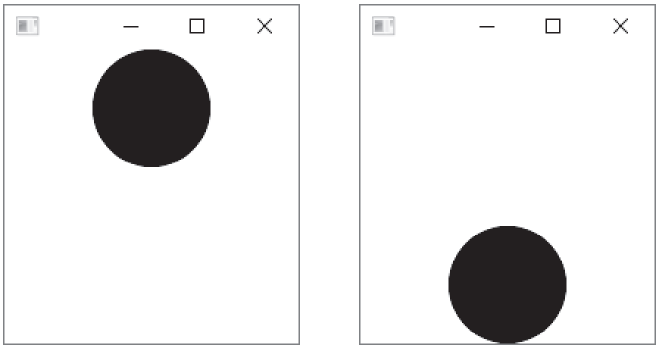
Code Listing 14-10 shows a complete program that animates a circle, moving it from the top of the application's window to the bottom. Figure 14-18 shows the program's output.

**Code Listing 14-10** `(BallDrop.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.layout.Pane;
import javafx.util.Duration;
import javafx.animation.TranslateTransition;

```

```
public class BallDrop extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 200.0;
      final double SCENE_HEIGHT = 200.0;

      // Constants for the ball
      final double START_X = 100.0, START_Y = 40.0;
      final double END_X = 100.0, END_Y = 160.0;
      final double RADIUS = 40.0;

      // Constant for the animation duration
      final double ONE_SEC = 1000.0;

      // Create the ball.
      Circle ball = new Circle(RADIUS);

      // Create the Transition object.
      TranslateTransition ttrans =
         new TranslateTransition(new Duration(ONE_SEC), ball);
      ttrans.setFromX(START_X);
      ttrans.setFromY(START_Y);
      ttrans.setToX(END_X);
      ttrans.setToY(END_Y);

      // Add the ball to a Pane.
      Pane pane = new Pane(ball);

      // Create a Scene and display it.
      Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
      primaryStage.setScene(scene);
      primaryStage.show();

      // Play the animation.
      ttrans.play();
   }
}
```

**Figure 14-18** Output of `BallDrop.java` (Oracle Corporate Counsel)



Position of the circle at the beginning of the animation

Position of the circle at the end of the animation

## The `RotateTransition` Class

The `RotateTransition` class creates an animation in which a node rotates. The `RotateTransition` class has three constructors, summarized in Table 14-13.

**Table 14-13** `RotateTransition` class constructors

| Constructor | Description |
|---|---|
| `RotateTransition()` | Creates an empty `RotateTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setNode` method to specify the node to animate. |
| `RotateTransition(`*`duration`*`)` | The *duration* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `RotateTransition` class's `setNode` method to specify the node to animate. |
| `RotateTransition(`*`duration`*`, `*`node`*`)` | The *duration* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *node* argument, an object of the `Node` class (or its subclasses) is the node to animate. |

The following code snippet shows an example of creating a `Line`, then creating a `RotateTransition` object to animate it. The duration of the animation is 5 seconds.

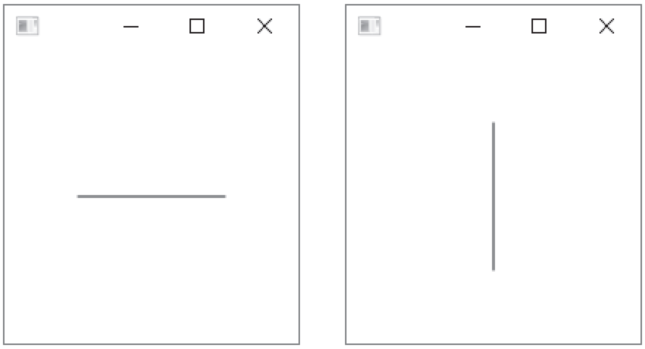
```
Line myLine = new Line(50, 100, 150, 100);
RotateTransition rtrans =
         new RotateTransition(new Duration(5000), myLine);
```

The `RotateTransition` class rotates a node around its center point, from its starting angle, to its ending angle. You use the `RotateTransition` class's `setFromAngle` and `setToAngle` methods to specify the starting angle (the “from” angle) and the ending angle (the “to” angle). You pass an angle, in degrees, to each of these methods. The following code snippet shows an example:

```
Line myLine = new Line(50, 100, 150, 100);
RotateTransition rtrans =
   new RotateTransition(new Duration(3000), myLine);
rtrans.setFromAngle(0.0);
rtrans.setToAngle(90.0);
rtrans.play();
```

This code creates a `Line`, then creates a `RotateTransition` object to animate it. The duration of the animation is 3 seconds. When the animation is played (by calling the `rtrans.play()` method), the `Line` object will be angled at 0 degrees, and it will rotate to the 90 degree position. It will take 3 seconds for the `Line` object to complete the rotation. Figure 14-19 shows the position of the line at the beginning and end of the animation.

**Figure 14-19** A line rotating from 0 degrees to 90 degrees (Oracle Corporate Counsel)



Position of the line at the beginning of the animation

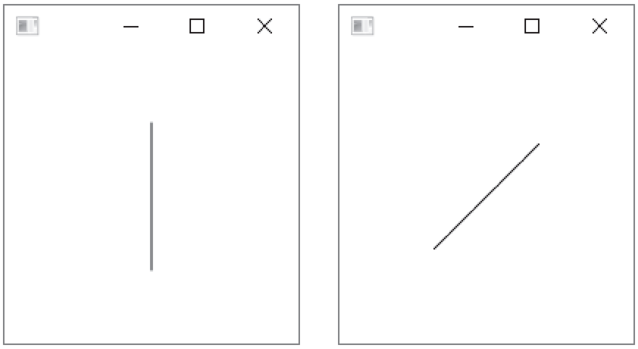
Position of the line at the end of the animation

Keep in mind the value you pass to the `setToAngle` method is the node’s ending angle. In the previous code snippet, the line rotates from the 0 degree position to the 180 degree position. If you prefer, you can call the `setByAngle` method to specify the amount of rotation. The following code snippet shows an example:

```
Line myLine = new Line(50, 100, 150, 100);
RotateTransition rtrans =
   new RotateTransition (new Duration(3000), myLine);
rtrans.setFromAngle(90.0);
rtrans.setByAngle(45.0);
rtrans.play();
```

In this example, the transition object initially positions the line at the 90 degree position, and rotates it by an additional 45 degrees. So, the ending angle will be 135 degrees. Figure 14-20 shows the position of the line at the beginning and end of the animation.

**Figure 14-20** A line starting at 90 degrees, rotating by an additional 45 degrees (Oracle Corporate Counsel)



Position of the line at the beginning of the animation

Position of the line at the end of the animation

**TIP:** If you pass a positive angle to the `setToAngle` or the `setByAngle` methods, the rotation will be clockwise. If you pass a negative angle to either of these methods, the rotation will be counterclockwise.

Code Listing 14-11 shows a complete program that animates an image of a poker card. When the user clicks a button, the image spins around 360 degrees. Figure 14-21 shows the program's output.

**Code Listing 14-11** `(RotateImage.java)`

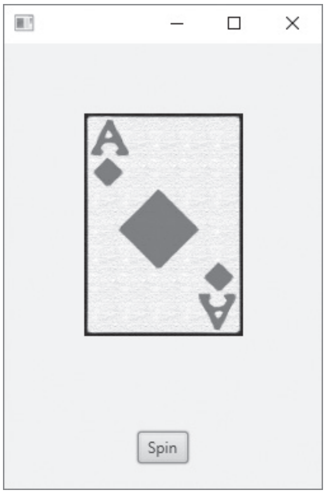
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.animation.RotateTransition;
import javafx.util.Duration;
import javafx.scene.control.Button;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

/**
  An Animation Demo
*/

public class RotateImage extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 250.0;
      final double SCENE_HEIGHT = 350.0;

      // Constants for the animation
      final double HALF_SEC = 500.0;
      final double FROM_ANGLE = 0.0;
      final double TO_ANGLE = 360.0;

      // Constant for the layout padding space
      final double PADDING_SPACE = 20.0;

      // Create a BorderPane.
      BorderPane borderPane = new BorderPane();

      // Create an Image and ImageView components.
      Image image = new Image("file:Ace_Diamonds.png");
      ImageView imageView = new ImageView(image);

      // Create a RotateTransition object for the ImageView.
      RotateTransition rtrans =
         new RotateTransition(new Duration(HALF_SEC), imageView);
      rtrans.setFromAngle(FROM_ANGLE);
      rtrans.setToAngle(TO_ANGLE);

      // Put the ImageView in the center of the BorderPane.
      borderPane.setCenter(imageView);

      // Create a Button to start the animation.
      Button spinButton = new Button("Spin");

      // Create an event handler for the Button.
      spinButton.setOnAction(event ->
      {
         rtrans.play();
      });

      // Put the Button in an HBox.
      HBox hbox = new HBox(spinButton);
      hbox.setAlignment(Pos.CENTER);
      hbox.setPadding(new Insets(PADDING_SPACE));
```

```
73
74         // Put the HBox in the bottom region of the BorderPane.
75         borderPane.setBottom(hbox);
76
77         // Create a Scene and display it.
78         Scene scene = new Scene(borderPane, SCENE_WIDTH, SCENE_HEIGHT);
79         primaryStage.setScene(scene);
80         primaryStage.show();
81      }
82   }
```

Let's take a closer look at the program.

- Lines 33 through 42 define various constants that are used in the program.
- Line 45 creates a `BorderPane` object, which will become the root node.
- Line 48 creates an `Image` component using the image of an Ace of Diamonds poker card, and line 49 creates an `ImageView` component to display the image.
- Lines 52 and 53 create a `RotateTransition` object for the animation. The duration of the animation is 500 milliseconds (half a second), and the `ImageView` component is the node that will be animated.
- Lines 54 and 55 set the starting angle to 0 degrees, and the ending angle to 360 degrees.
- Line 58 adds the `ImageView` component to the `BorderPane`'s center region.
- Line 61 creates a Button control named `spinButton`.
- Lines 64 through 67 use a lambda expression to register an event listener for the Button. The event listener plays the animation.
- Lines 70 through 72 put the `spinButton` control in an `HBox`, with center alignment and padding of 20 pixels.
- Line 75 adds the `HBox` to the `BorderPane`'s bottom region.
- Line 78 creates a scene with the `BorderPane` as the root node.
- Line 79 sets the scene to the stage, and line 80 displays the application's window.

**Figure 14-21** Output of `RotateImage.java` (Oracle Corporate Counsel)



## The `ScaleTransition` Class

The `ScaleTransition` class creates an animation in which a node becomes larger or smaller over time. The `ScaleTransition` class has three constructors, summarized in Table 14-14.

**Table 14-14** `ScaleTransition` class constructors

| Constructor | Description |
|---|---|
| `ScaleTransition()` | Creates an empty `ScaleTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setNode` method to specify the node to animate. |
| `ScaleTransition(`*`duration`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `ScaleTransition` class's `setNode` method to specify the node to animate. |
| `ScaleTransition(`*`duration, node`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`node`* argument, an object of the `Node` class (or its subclasses) is the node to animate. |

To animate a node with the `ScaleTransition` class, you specify starting scale factors along the *X* and *Y* axes, and ending scale factors along the *X* and *Y* axes. When the animation is played, the transition class will change the size of the node from the starting scale factors to the ending scale factors, over a specified duration of time. The following code snippet shows an example:

```
Circle myCircle = new Circle(50, 50, 10);
ScaleTransition strans =
         new ScaleTransition(new Duration(2000), myCircle);
strans.setFromX(1.0);
strans.setFromY(1.0);
strans.setToX(3.0);
strans.setToY(3.0);
strans.play();
```

This code creates a `Circle`, with its center point at (50, 50) and a radius of 10. It then creates a `ScaleTransition` object to animate the `Circle` object. The duration of the animation is 2 seconds. When the animation is played, the `Circle` object will be displayed with a starting scale factor of 1 (along both the *X* and *Y* axes), and it will increase in size to a scale factor of 3 (along both the *X* and *Y* axes). It will take 2 seconds for the animation to complete.

Code Listing 14-12 shows a complete example. The program displays a `Text` object with a starting scale factor of 1 (along both axes) and over a duration of 5 seconds, it increases the `Text` object's size until it has an ending scale factor of 5. The program's output is shown in Figure 14-22.

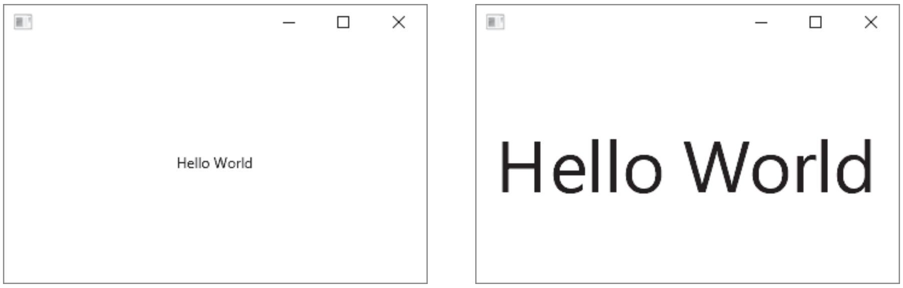
**Code Listing 14-12** `(ScaleTransitionText.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.geometry.Pos;
import javafx.scene.text.Text;
import javafx.util.Duration;
import javafx.animation.ScaleTransition;

public class ScaleTransitionText extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 350.0;
      final double SCENE_HEIGHT = 200.0;

      // Constants for the transition
      final double FIVE_SEC = 5000.0;
      final double START_SCALE = 1.0;
      final double END_SCALE = 5.0;

      // Create a Text object.
      Text text = new Text("Hello World");

      // Create the Transition object.
      ScaleTransition strans =
         new ScaleTransition(new Duration(FIVE_SEC), text);
      strans.setFromX(START_SCALE);
      strans.setFromY(START_SCALE);
      strans.setToX(END_SCALE);
      strans.setToY(END_SCALE);

40          // Add the text to an HBox.
41          HBox hbox = new HBox(text);
42          hbox.setAlignment(Pos.CENTER);
43
44          // Create a Scene and display it.
45          Scene scene = new Scene(hbox, SCENE_WIDTH, SCENE_HEIGHT);
46          primaryStage.setScene(scene);
47          primaryStage.show();
48
49          // Play the animation.
50          strans.play();
51       }
52    }
```

Let's take a closer look at the program.

- Lines 21 through 27 define various constants.
- Line 30 creates a `Text` object that will display "Hello World". Notice we do not specify an (*X*, *Y*) location for the text. This is okay, because we are going to put the `Text` object in an `HBox`. We will let the `HBox` position the text.
- Lines 33 and 34 create a `ScaleTransition` object to animate the `Text` object. The duration of the animation is 5 seconds.
- Line 35 sets the starting scale factor for the *X* axis to 1, and line 36 sets the starting scale factor for the *Y* axis to 1.
- Line 37 sets the ending scale factor for the *X* axis to 5, and line 38 sets the ending scale factor for the *Y* axis to 5.
- Line 41 adds the `Text` object to an `HBox`, and line 42 sets the `HBox`'s alignment to center.
- Lines 45 through 47 create the scene, add the `HBox` to the scene as the root node, sets the scene to the stage, and displays the window.
- Line 50 plays the animation.

**Figure 14-22** An animation in which text is scaled to 5 times its original size (Oracle Corporate Counsel)



Hello World

Hello World

Size of the text at the beginning of the animation

Size of the text at the end of the animation

## The `StrokeTransition` Class

The `StrokeTransition` class creates an animation in which a shape's stroke color changes over a period of time. (Note the `StrokeTransition` class works only with objects of the `Shape` class, or one of its subclasses.) The `StrokeTransition` class has five constructors, summarized in Table 14-15.

**Table 14-15** `StrokeTransition` class constructors

| Constructor | Description |
|---|---|
| `StrokeTransition()` | Creates an empty `StrokeTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setShape` method to specify the shape to animate, the `setFromValue` method to specify the starting color, and the `setToValue` method to specify the ending color. |
| `Strokeiransition(`*`duration`*`)` | The *duration* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `StrokeTransition` class's `setShape` method to specify the shape to animate, the `setFromValue` method to specify the starting color, and the `setToValue` method to specify the ending color. |
| `StrokeTransition (`*`duration, shape`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`shape`* argument, an object of the `Shape` class (or its subclasses) is the node to animate. Call the `setFromValue` method to specify the starting color, and the `setToValue` method to specify the ending color. |
| `StrokeTransition (`*`duration, fromValue, toValue`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`fromValue`* argument, an object of the `Color` class, specifies the starting color. The *`toValue`* argument, an object of the `Color` class, specifies the ending color. Call the `setShape` method to specify the shape to animate. |
| `StrokeTransition (`*`duration, shape, fromValue, toValue`*`)` | The *dura*tion argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`shape`* argument, an object of the `Shape` class (or its subclasses) is the node to animate. The *`fromValue`* argument, an object of the `Color` class, specifies the starting color. The *`toValue`* argument, an object of the `Color` class, specifies the ending color. Call the `setShape` method to specify the shape to animate. |

To animate a shape with the `StrokeTransition` class, you specify a starting color and an ending color. When the animation is played, the transition class will gradually change the shape's stroke color from the starting color to the ending color, over a specified duration of time. The following code snippet shows an example:

```
Circle myCircle = new Circle(50, 50, 10);
StrokeTransition strtrans =
      new StrokeTransition(new Duration(2000), myCircle,
                           Color.YELLOW, Color.GREEN);
strtrans.play();
```

This code creates a `Circle`, with its center point at (50, 50) and a radius of 10. It then creates a `StrokeTransition` object to animate the `Circle` object. The duration of the animation is 2 seconds. When the animation is played, the `Circle` object will be displayed with a starting stroke color of yellow, and it will gradually change the stroke color to green. It will take 2 seconds for the animation to complete.

## The `FillTransition` Class

The `FillTransition` class creates an animation in which a shape's fill color changes over a period of time. (Note the `FillTransition` class works only with objects of the `Shape` class, or one of its subclasses.) The `FillTransition` class has five constructors, summarized in Table 14-16.

**Table 14-16** `FillTransition` class constructors

| Constructor | Description |
|---|---|
| `FillTransition()` | Creates an empty `FillTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setShape` method to specify the shape to animate, the `setFromValue` method to specify the starting color, and the `setToValue` method to specify the ending color. |
| `FillTransition(`*`duration`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `FillTransition` class's `setShape` method to specify the shape to animate, the `setFromValue` method to specify the starting color, and the `setToValue` method to specify the ending color. |
| `FillTransition(`*`duration`*`, `*`shape`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`shape`* argument, an object of the `Shape` class (or its subclasses) is the node to animate. Call the `setFromValue` method to specify the starting color, and the `setToValue` method to specify the ending color. |
| `FillTransition(`*`duration`*`, `*`fromValue`*`, `*`toValue`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`fromValue`* argument, an object of the `Color` class, specifies the starting color. The *`toValue`* argument, an object of the `Color` class, specifies the ending color. Call the `setShape` method to specify the shape to animate. |

To animate a shape with the `FillTransition` class, you specify a starting color and an ending color. When the animation is played, the transition class will gradually change the shape's fill color from the starting color to the ending color, over a specified duration of time. The following code snippet shows an example:

```
Circle myCircle = new Circle(50, 50, 10);
FillTransition ftrans =
    new FillTransition (new Duration(2000), myCircle,
                        Color.LIGHTBLUE, Color.DARKBLUE);
ftrans.play();
```

This code creates a `Circle`, with its center point at (50, 50) and a radius of 10. It then creates a `FillTransition` object to animate the `Circle` object. The duration of the animation is 2 seconds. When the animation is played, the `Circle` object will be displayed with a starting fill color of light blue, and it will gradually change the fill color to dark blue. It will take 2 seconds for the animation to complete.

## The `FadeTransition` Class

The `FadeTransition` class creates an animation in which a node fades in or out over a period of time. The `FadeTransition` class has three constructors, summarized in Table 14-17.

**Table 14-17** `FadeTransition` class constructors

| Constructor | Description |
|---|---|
| `FadeTransition()` | Creates an empty `FadeTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setNode` method to specify the node to animate. |
| `FadeTransition(`*`duration`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `ScaleTransition` class's `setNode` method to specify the node to animate. |
| `FadeTransition(`*`duration`*`, `*`node`*`)` | The *`duration`* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *`node`* argument, an object of the `Node` class (or its subclasses) is the node to animate. |

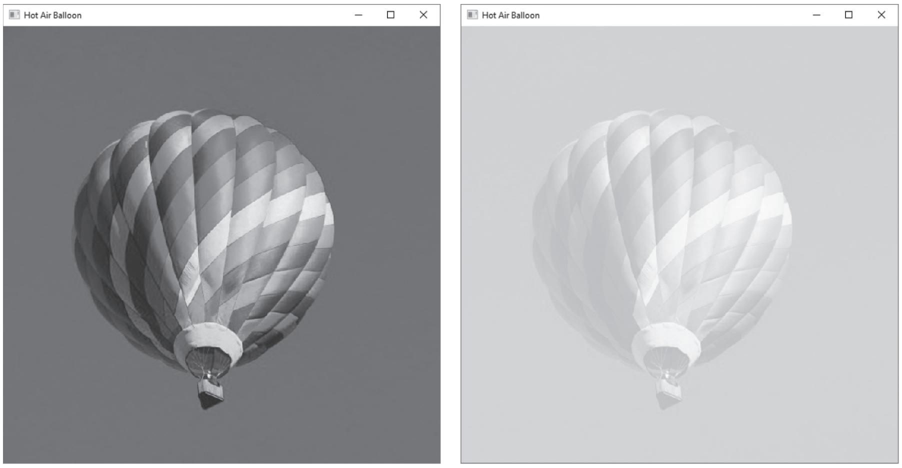
All nodes have an *opacity* value that determines whether the node is transparent or opaque. An opacity value is a `double` in the range of 0 through 1. When a node's opacity value is 0, the node is completely transparent. When a node's opacity value is 1, the node is completely opaque (you cannot see through it). When a node's opacity value is between 0 and 1, the node is semitransparent.

To animate a node with the `FadeTransition` class, you specify values for starting opacity and ending opacity. When the animation is played, the transition class will gradually change the node's opacity from the specified starting value to the specified ending value. The following code snippet shows an example:

```
// Create an Image and an ImageView.
Image image = new Image("file:HotAirBalloon.jpg");
ImageView imageView = new ImageView(image);

// Create a FadeTransition to animate the ImageView.
FadeTransition ftrans =
    new FadeTransition(new Duration(1000), imageView);
ftrans.setFromValue(1.0);
ftrans.setToValue(0.25);
ftrans.play();
```

This code creates an `Image` component and an `ImageView` component to display an image, then creates a `FadeTransition` object to animate the `ImageView` object. The duration of the animation is 1 second. When the animation is played, the image will be displayed with an opacity of 1.0 (it will be completely opaque). As the animation plays, the image's opacity will gradually decrease to 0.25 (twenty-five percent opaque). Figure 14-23 shows an example of the code's output at the beginning and the end of the animation.

**Figure 14-23** Fade transition example



The image at the beginning of the animation (opacity = 1.0)

The image at the end of the animation (opacity = 0.25)

## Controlling the Animation

Each of the transition classes we have discussed are subclasses of the `Animation` class and the `Transition` class. Table 14-18 summarizes a few of the methods inherited from the `Animation` class, that give you greater control over the animation.

**Table 14-18** A few methods inherited from the `Animation` class

| Method | Description |
|---|---|
| `jumpTo(`*`time`*`)` | The *`time`* argument is a `Duration` object. This method causes the animation to jump to the time position specified by *`time`*. (For example, if the *`time`* argument specifies 5 seconds, the method will jump the animation to the 5-second position.) |
| `pause()` | Pauses the animation. (This method has no effect if the animation is not playing.) |
| `play()` | Plays the animation from its current position. (This method has no effect if the animation is already running.) |
| `playFrom(`*`time`*`)` | The *`time`* argument is a `Duration` object. This method plays the animation from the time position specified by *`time`*. (For example, if the animation lasts 6 seconds, and the *`time`* argument specifies 3 seconds, the method will play the animation from the 3-second position.) |
| `playFromStart()` | Plays the animation from its beginning. |
| `setCycleCount(`*`value`*`)` | The *`value`* argument is an integer. Defines the number of times the animation will repeat. If you want the animation to repeat indefinitely, specify the value `Animation.INDEFINITE` as the argument. |
| `setAutoReverse(`*`value`*`)` | The *`value`* argument is a Boolean. If *`value`* is `true`, then the animation will play in reverse on every other cycle. If *`value`* is `false`, then the animation will play normally. |
| `stop()` | Stops the animation, and resets the current position to the beginning of the animation. |

## Specifying an Interpolator

All of the transition classes inherit a method named `setInterpolator` from the `Transition` class. The `setInterpolator` method allows you to specify how the animation begins and ends. The general format for the method is

```
transitionObject.setInterpolator(interpolatorValue);
```

where *`transitionObject`* is a transition object, and *`interpolatorValue`* is usually one of the constants listed in Table 14-19. (The `Interpolator` class is in the `javafx.animation.Interpolator` package.)

**Table 14-19** `Interpolator` constants

| Method | Description |
|---|---|
| `Interpolator.EASE_BOTH` | This interpolator causes the animation to ease in and ease out. In other words, the animation starts slowly, accelerates, then slows down at the end. This is the default interpolator. |
| `Interpolator.EASE_IN` | This interpolator causes the animation to ease in (start slowly), but stop quickly. |
| `Interpolator.EASE_OUT` | This interpolator causes the animation to start quickly, but ease out (gradually slow down at the end). |
| `Interpolator.DISCRETE` | With this interpolator, the animation appears to jump from its beginning immediately to its end. |
| `Interpolator.LINEAR` | This interpolator causes the animated object to change its affected property (position, scale, rotation angle, color, and so forth) at a constant rate over time. |

The following code snippet shows an example:

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
         new TranslateTransition(new Duration(3000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(50);
ttrans.setToX(100);
ttrans.setToY(50);
ttrans.setInterpolator(Interpolator.EASE_IN);
ttrans.play();
```

This code creates a `Circle`, then creates a `TranslateTransition` object to animate it. The duration of the animation is 3 seconds. When the animation is played, the `Circle` object will start at (0, 50), and it will move to the location (100, 50). It will take 3 seconds for the `Circle` object to complete the move. Because the interpolator is set to `Interpolator.EASE_IN`, the animation will start slowly, but end abruptly.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.13 Which transition class causes a node to move from one position on the screen to another?

14.14 Which transition class causes a node to rotate?

14.15 Which transition class causes a node to become larger or smaller?

14.16 Which transition class causes a node's stroke color to change?

14.17 Which transition class causes a node's fill color to change?

14.18 Which transition class causes a node to fade?

14.19 How do you specify the amount of time that an animation should last?

14.20 In what package is the `Duration` class?

14.21 How can you change an animation's interpolator?

## 14.3 Effects

**CONCEPT: JavaFX provides several classes that can be used to apply special effects to nodes in the scene graph.**

VideoNote
JavaFX Effects

JavaFX provides several classes you can use to create special visual effects for the nodes in a scene graph. For example, you can give a node a 3D appearance by displaying a drop shadow or an inner shadow with it. You can also filter images by adding effects such as blurs and color adjustments. Table 14-20 lists several of the effect classes, each of which is in the `javafx.scene.effect` package.

**Table 14-20** Some of the effect classes

| Effect Class | Description |
|---|---|
| `DropShadow` | Creates a drop shadow, which is a shadow that appears behind a node. You specify a color, radius, and offset for the drop shadow. |
| `InnerShadow` | Creates a shadow that appears inside the edges of a node, thus giving the node a recessed appearance. You specify a color, radius, and offset for the shadow. |
| `ColorAdjust` | Allows you to adjust an image's hue, saturation, brightness, and contrast. |
| `BoxBlur` | Creates a blurring effect using the box blur algorithm. |
| `GaussianBlur` | Creates a blurring effect using the Gaussian algorithm. |
| `MotionBlur` | Blurs the pixels of an image in a specific direction, creating the illusion of motion. |
| `SepiaTone` | Gives an image an antique, sepia tone effect |
| `Glow` | Creates a glowing effect |
| `Reflection` | Creates a reflection of a node |

To create a special effect for a node, you create an object of the desired effect class, then call the node's `setEffect` method, passing the effect object as an argument. Let's look at how this relates to each of the effect classes listed in Table 14-20.

### The DropShadow Class

Use the `DropShadow` class to create a shadow behind a node, giving it the appearance of being raised above a surface. Code Listing 14-13 demonstrates how a drop shadow can be applied to a circle.

**Code Listing 14-13** `(DropShadowDemo.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
```

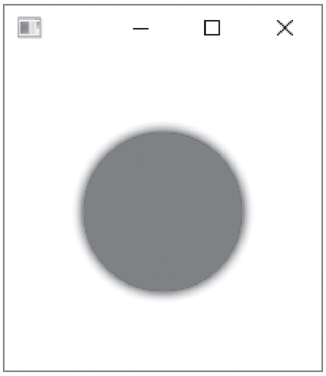
```
 5 import javafx.scene.shape.Circle;
 6 import javafx.scene.paint.Color;
 7 import javafx.scene.effect.DropShadow;
 8
 9 public class DropShadowDemo extends Application
10 {
11    public static void main(String[] args)
12    {
13       launch(args);
14    }
15
16    @Override
17    public void start(Stage primaryStage)
18    {
19       // Constants for the scene size
20       final double SCENE_WIDTH = 200.0;
21       final double SCENE_HEIGHT = 200.0;
22
23       // Constants for the Circle
24       final double X = 100.0, Y = 100.0, RADIUS = 50.0;
25
26       // Create a DropShadow
27       DropShadow dropShadow = new DropShadow();
28
29       // Create a Circle object.
30       Circle myCircle = new Circle(X, Y, RADIUS, Color.RED);
31       myCircle.setEffect(dropShadow);
32
33       // Add the Circle to a Pane.
34       Pane pane = new Pane(myCircle);
35
36       // Create a Scene and display it.
37       Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
38       primaryStage.setScene(scene);
39       primaryStage.show();
40    }
41 }
```

Here are the lines of code to take note of:

- Line 27 creates a `DropShadow` object named `dropShadow`, using the `DropShadow` class's no-arg constructor.
- Line 30 creates a `Circle` object.
- Line 31 calls the `Circle` object's `setEffect` method, passing the `dropShadow` object as an argument.

The program's output is shown in Figure 14-24.

**Figure 14-24** Output of DropShadowDemo.java (Oracle Corporate Counsel)



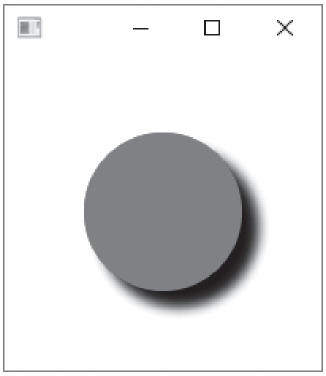
If you are not satisfied with the appearance of the default drop shadow, you can use the `DropShadow` class methods listed in Table 14-21 to alter its appearance. For example, the following code snippet (taken from *DropShadowDemo2.java*) displays a circle with a drop shadow that has *X* and *Y* offsets of 10. As a result, the shadow is shifted 10 pixels to the right, and 10 pixels down. The code's output is shown in Figure 14-25.

```
// Create a DropShadow
DropShadow dropShadow = new DropShadow();
dropShadow.setOffsetX(10);
dropShadow.setOffsetY(10);

// Create a Circle object.
Circle myCircle = new Circle(X, Y, RADIUS, Color.RED);
myCircle.setEffect(dropShadow);
```

**Table 14-21** Some of the DropShadow class methods

| Method | Description |
|---|---|
| `setColor(`*`color`*`)` | The argument is a `Color` object that specifies the color of the shadow. The default value is `Color.BLACK`. |
| `setRadius(`*`value`*`)` | The argument is a `double` that specifies the radius of the shadow's blur. The default value is 10.0. |
| `setSpread(`*`value`*`)` | The argument is a `double` between 0.0 and 1.0 that specifies how far the shadow spreads before it starts to blur. The default value is 0.0. |
| `setOffsetX(`*`value`*`)` | The argument is a `double` that specifies the offset of the shadow in the *X* direction. The default value is 0.0. |
| `setOffsetY(`*`value`*`)` | The argument is a `double` that specifies the offset of the shadow in the *Y* direction. The default value is 0.0. |
| `setBlurType(`*`blurType`*`)` | The argument is a `BlurType` enum constant (from the `javafx.scene.effect` package) that specifies the algorithm used to blur the shadow. The possible values are `BlurType.GAUSSIAN`, `BlurType.ONE_PASS_BOX`, `BlurType.TWO_PASS_BOX`, and `BlurType.THREE_PASS_BOX` (the default value). |

**Figure 14-25** Drop shadow with *X* and *Y* offsets of 10 (Oracle Corporate Counsel)



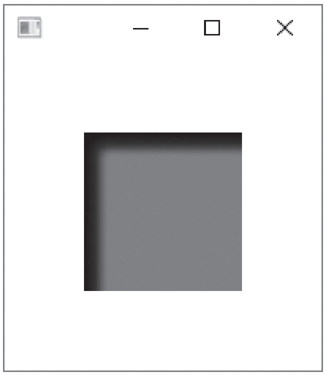
## The `InnerShadow` Class

Use the `InnerShadow` class to create a shadow on the inside edge of a node, giving it a recessed appearance. The following code snippet, taken from *InnerShadowDemo.java*, demonstrates how an inner shadow can be applied to a rectangle. The code's output is shown in Figure 14-26.

```
// Create an InnerShadow
InnerShadow innerShadow = new InnerShadow();
innerShadow.setOffsetX(10);
innerShadow.setOffsetY(10);

// Create a Rectangle object.
Rectangle box = new Rectangle(X, Y, WIDTH, HEIGHT);
box.setFill(Color.RED);
box.setEffect(innerShadow);
```

Notice in the code snippet we call the `InnerShadow` class's `setOffsetX` and `setOffsetY` methods to shift the shadow. The `InnerShadow` class provides all of the methods that are listed for the `DropShadow` class in Table 14-21 except the `setSpread` method. Instead, the `InnerShadow` class provides a method named `setChoke` that specifies how far the shadow spreads before it starts to blur. The default value is 0.0. The argument is a `double` between 0.0 and 1.0.

**Figure 14-26** A rectangle with an inner shadow (Oracle Corporate Counsel)



## The `ColorAdjust` Class

The `ColorAdjust` class allows you to adjust a node's hue, saturation, brightness, and contrast, at the pixel level. You can use the methods listed in Table 14-22 to make these adjustments.

**Table 14-22** Some of the `ColorAdjust` class methods

| Method | Description |
|---|---|
| `setHue(`*`value`*`)` | This method adjusts the hue, or color, of each pixel in an image, shape, or other node. The argument, a `double` in the range of −1.0 through 1.0, `represents` the amount to change the color. The default value is 0.0. |
| `setSaturation(`*`value`*`)` | This method adjusts the saturation, or intensity, of each pixel in an image, shape, or other node. The argument, a `double` in the range of −1.0 through 1.0, represents the amount to change the saturation. The default value is 0.0. |
| `setBrightness(`*`value`*`)` | This method adjusts the brightness of each pixel in an image, shape, or other node. The argument, a `double` in the range of −1.0 through 1.0, represents the amount to change the brightness. The default value is 0.0. |
| `setContrast(`*`value`*`)` | This method adjusts the `contrast`, or the difference between bright areas and dark areas, in an image, shape, or other node. The argument, a `double` in the range of −1.0 through 1.0, represents the amount to change the color. The default value is 0.0. |

The following code snippet shows an example of using each of these methods to manipulate an image:

```
ColorAdjust colorAdjust = new ColorAdjust();
colorAdjust.setHue(0.25);
colorAdjust.setSaturation(0.5);
colorAdjust.setBrightness(-0.25);
colorAdjust.setContrast(0.1);

Image image = new Image("file:flower.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(colorAdjust);
```

## The `BoxBlur`, `GaussianBlur`, and `MotionBlur` Classes

Each of these classes can be used to make a node look blurry, or out of focus. The `BoxBlur` class uses an algorithm in which a pixel's value is recalculated as the average of the values of neighboring pixels within an imaginary box. The `GaussianBlur` class tends to produce a smoother blur than the `BoxBlur` class, but takes more processing time to create. The `MotionBlur` class blurs pixels in a specific direction, to create the illusion of motion. Let's look at each of these classes.

### The `BoxBlur` Class

The following code snippet shows an example of how to use the `BoxBlur` class to blur the text produced by a `Text` object. In Figure 14-27, the text on the right is the text produced by this code snippet. The text on the left is the same text, without the blur effect.

```
// Make a BoxBlur.
BoxBlur boxBlur = new BoxBlur();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(boxBlur);
```

**Figure 14-27** Comparison of normal text and text with a `BoxBlur` effect

Hello World
Hello World

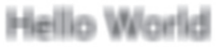
You can increase the smoothness of the `BoxBlur` effect by increasing its number of iterations. By default, the class makes only one pass over the image. Each time it makes an additional pass, however, the blurring effect becomes smoother. You can call the class's `setIterations` method, passing an integer argument, to set the number of iterations. The maximum number of iterations is 3. Here is an example code snippet:

```
BoxBlur boxBlur = new BoxBlur();
boxBlur.setIterations(3);
```

If you want to increase or decrease the amount of blurriness produced by the `BoxBlur` class, you can use the `setWidth` and `setHeight` methods, passing a `double` argument, to change the size of the imaginary box that is used for recalculating each pixel's value. By default, the box is 5 pixels wide and 5 pixels high. The larger the box, the blurrier the effect. The minimum width and height is 1 pixel (which produces no effect at all) and the maximum width and height is 255 pixels. The following code snippet sets the width and height to 10 pixels. The code's output is shown in Figure 14-28.

```
// Make a BoxBlur.
BoxBlur boxBlur = new BoxBlur();
boxBlur.setWidth(10.0);
boxBlur.setHeight(10.0);

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(boxBlur);
```

**Figure 14-28** `BlurBox` width and height set to 10 pixels



### The `GaussianBlur` Class

The following code snippet shows an example of how to use the `GaussianBlur` class to blur the text produced by a `Text` object. In Figure 14-29, the text on the right is the text that is produced by this code snippet. The text on the left is the same text, without the blur effect.

```
// Make a GaussianBlur.
GaussianBlur gaussianBlur = new GaussianBlur();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(gaussianBlur);
```

**Figure 14-29** Comparison of normal text and text with a `GaussianBlur` effect

Hello World
Hello World

You can also increase or decrease the amount of blurriness produced by the `GaussianBlur` class by changing its blur radius. You call the `setRadius` method, passing the desired radius as a `double`. The larger the radius, the blurrier the effect. By default, the radius is 10.0. The minimum radius is 0.0 (which produces no effect at all) and the maximum radius is 63.0. Here is an example code snippet that sets the blur radius to 20.0:

```
GaussianBlur gaussianBlur = new GaussianBlur();
gaussianBlur.setRadius(20.0);
```

### The `MotionBlur` Class

The following code snippet shows an example of how to use the `MotionBlur` class to blur the text produced by a `Text` object. In Figure 14-30, the text on the right is the text that is produced by this code snippet. The text on the left is the same text, without the blur effect.

```
// Make a MotionBlur.
MotionBlur motionBlur = new MotionBlur();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(motionBlur);
```

**Figure 14-30** Comparison of normal text and text with a `MotionBlur` effect

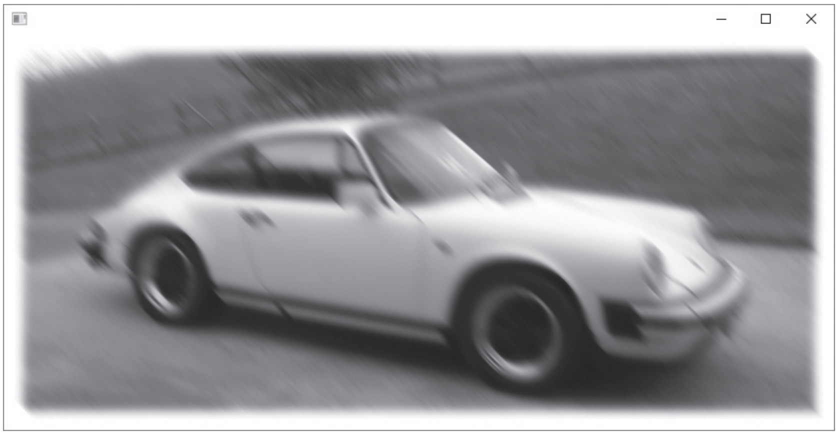
Hello World
Hello World

The `MotionBlur` class blurs pixels in a particular direction. By default, the direction is 0.0 degrees, but you can change that by calling the class's `setAngle` method, passing the desired angle as a `double`. You can also increase or decrease the amount of blurriness produced by the `MotionBlur` class by changing its blur radius. You call the `setRadius` method, passing the desired radius as a `double`. The larger the radius, the blurrier the effect. By default, the radius is 10.0. The minimum radius is 0.0 (which produces no effect at all) and the maximum radius is 63.0.

The following code snippet creates a `MotionBlur` effect with a blur angle of 45 degrees, and a blur radius of 20.0. The effect is then applied to an `ImageView` object. Figure 14-31 shows the code's output.

```
// Make a MotionBlur.
MotionBlur motionBlur = new MotionBlur();
motionBlur.setAngle(45.0);
motionBlur.setRadius(20.0);

// Apply the effect to an image.
Image image = new Image("file:Porsche.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(motionBlur);
```

**Figure 14-31** Motion blur applied to an image



## The `SepiaTone` Class

You can use the `SepiaTone` class to give an image an antique, sepia tone appearance. The following code snippet applies the effect to an image:

```
// Make a SepiaTone effect.
SepiaTone sepia = new SepiaTone();

// Apply the effect to an image.
Image image = new Image("file:Ship.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(sepia);
```

You can change the intensity of the `SepiaTone` class's effect by calling the `setLevel` method, passing a `double` in the range of 0.0 through 1.0. The higher the value, the more intense the effect. By default, the level is 1.0. Here is an example code snippet that sets the level to 0.5:

```
SepiaTone sepia = new SepiaTone();
sepia.setLevel(0.5);
```

## The `Glow` Class

You can use the `Glow` class to produce a glowing effect. The following code snippet applies the effect to an image:

```
// Make a Glow effect.
Glow glow = new Glow();

// Apply the effect to an image.
Image image = new Image("file:SanDiego.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(glow);
```

You can change the intensity of the `Glow` class's effect by calling the `setLevel` method, passing a `double` in the range of 0.0 through 1.0. The higher the value, the more intense the effect. By default, the level is 0.3. Here is an example code snippet that sets the level to 1.0:

```
Glow glow = new Glow();
glow.setLevel(0.5);
```

Figure 14-32 shows how an image appears with and without the `Glow` effect. The image on the left is the original image, without the `Glow` effect, and the image on the right has a `Glow` effect applied, with the level set to 1.0.

**Figure 14-32** An image without, and with, the `Glow` effect

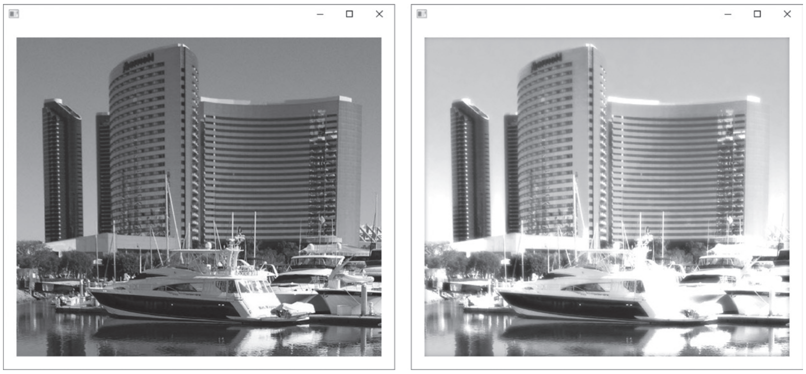


Image without the `Glow` effect

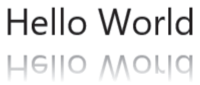
Image with the `Glow` effect

## The `Reflection` Class

You can use the `Reflection` class to create a reflection of a node. The reflection appears below the node. The following code snippet applies the effect to a `Text` object. The code's output is shown in Figure 14-33.

```
// Make a Reflection effect.
Reflection reflect = new Reflection();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(reflect);
```

**Figure 14-33** The `Reflection` effect applied to a `Text` object



The `Reflection` class provides several methods, listed in Table 14-23, for fine-tuning the way the reflection appears.

**Table 14-23** Some of the `Reflection` class methods

| Method | Description |
|---|---|
| `setTopOffset(`*`value`*`)` | The *`value`* argument is a `double`. This method sets *`value`* as the distance from the bottom of the node and the top of the reflection. The default value is 0.0. |
| `setFraction(`*`value`*`)` | The *`value`* argument is a `double`. This method sets *`value`* as the fraction of the node that appears in the reflection. The default value is 0.75, which means that the bottom 75 percent of the node will appear in the reflection. |
| `setTopOpacity(`*`value`*`)` | The *`value`* argument is a `double` in the range of 0.0 through 1.0. This method sets *`value`* as the opacity of the reflection at its top edge, with 0.0 being fully transparent, and 1.0 being fully opaque. The default value is 0.5. |
| `setBottomOpacity(`*`value`*`)` | The *`value`* argument is a `double` in the range of 0.0 through 1.0. This method sets *`value`* as the opacity of the reflection at its bottom edge, with 0.0 being fully transparent, and 1.0 being fully opaque. The default value is 0.0. |

## Combining Effects

The `Node` class's `setEffect` method accepts only one effect object as an argument. If you call the `setEffect` method more than once for a particular node, the last effect object that you pass as an argument replaces any previously passed effect objects for that node. However, it is possible to combine effects using the `Effect` class's `setInput` method. The following code snippet shows an example of combining a `DropShadow` with a `Reflection`, then applying them both to a `Text` object:

```
// Create a DropShadow effect
DropShadow dropShadow = new DropShadow();

// Create a Reflection effect.
Reflection reflect = new Reflection();

// Combine the Reflection with the DropShadow.
reflect.setInput(dropShadow);

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(reflect);
```

The `setInput` method “chains” effects together. When this code executes, the drop shadow effect is applied first, then the reflection effect is applied. The results are shown in Figure 14-34. The following code snippet shows an example of chaining three effects together: a `DropShadow`, a `SepiaTone`, and a `Reflection`. The results are shown in Figure 14-35. (The sepia tone effect is not noticeable when the image is printed in black and white, but you can try this code for yourself to see the true result.)

```
// Make a DropShadow effect.
DropShadow dropShadow = new DropShadow();

// Make a SepiaTone effect.
SepiaTone sepia = new SepiaTone();

// Make a Reflection effect.
Reflection reflect = new Reflection();

// Combine the DropShadow with the SepiaTone.
sepia.setInput(dropShadow);

// Combine the SepiaTone with the Reflection.
reflect.setInput(sepia);

// Apply the effects to an image.
Image image = new Image("file:Ship2.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(reflect);
```

**Figure 14-34** Applying `DropShadow` and `Reflection` effects to a `Text` object

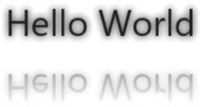
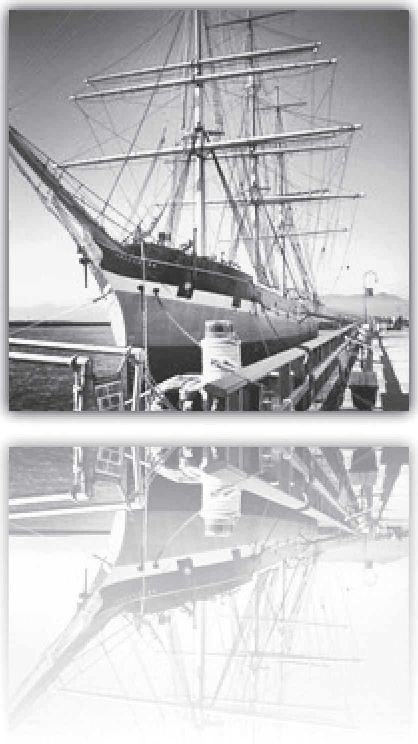


**Figure 14-35** Applying `DropShadow`, `sepiaTone`, and `Reflection` effects to an image



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

14.22 In what package are the effect classes?

14.23 Which effect class do you use to create a drop shadow?

14.24 Which effect class do you use to create a shadow inside the edges of a node?

14.25 Which effect class do you use to adjust an image's hue, saturation, brightness, and contrast?

14.26 What three effect classes can you use to create blur effects?

14.27 Which effect class do you use to create an antique sepia tone effect?

14.28 Which effect class do you use to create a glowing effect?

14.29 Which effect class do you use to create a reflection of a node?

14.30 If you want to combine multiple effects on a single node, which `Node` class method would you use to set the effects?

## 14.4 Playing Sound Files

**CONCEPT:** **In JavaFX, you use the `Media` and `MediaPlayer` classes to play sound files. The `Media` class loads a sound file into memory, and the `MediaPlayer` class provides methods for playing the sound file.**

VideoNote
Playing Sound Files with JavaFX

You can use JavaFX to play audio that is stored in one of the popular file formats .AIF, .MP3, or .WAV. Playing a sound file in a JavaFX application is a two-step process: First, you load the sound file into memory, and second, you play the sound file. This requires two classes from the JavaFX library: `Media` and `MediaPlayer`. Both of these classes are in the `javafx.scene.media` package.

The `Media` class can load an audio file from the computer's local file system, or from an Internet location. To load an audio file, create an instance of the `Media` class, passing the constructor a *uniform resource locator*, or *URI*, that specifies the file's name and location. A simple way to get the URI for a file is to use the `File` class from the `java.io package`. The following code snippet demonstrates how we can use the `File` class and the `Media` class to load a sound file named *guitar.wav*:

```
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
```

In this code snippet, the first statement creates a `File` object named `soundFile`, representing the *guitar.wav* file. The second statement creates a `Media` object. Notice the argument we are passing to the `Media` class constructor. The expression `soundFile.toURI().toString()` gives us a string version of the file's URI.

Because no path for the *guitar.wav* file was given in the first statement, it is assumed that the file is in the same directory or folder as the program's executable .class file. If you want to load a file from a different location, you can specify a path. Here is an example:

```
File soundFile = new File("C:\\media\\guitar.wav");
Media media = new Media(soundFile.toURI().toString());
```

Next, you create an instance of the `MediaPlayer` class, passing a reference to the `Media` object as an argument to the constructor. Here is an example:

```
MediaPlayer player = new MediaPlayer(media);
```

Once you have successfully created a `MediaPlayer` object, you can use the `MediaPlayer` methods listed in Table 14-24 to control the way the sound file is played.

**Table 14-24** Some of the `MediaPlayer` methods

| Method | Description |
|---|---|
| `setAutoPlay(`*`value`*`)` | The argument is a `boolean` value. If you pass `true` to the method, the sound file will be immediately played once the `MediaPlayer` object is created and ready. |
| `play()` | Starts playing the sound file. Normally, this method plays the sound file from its beginning, but if the sound file is currently paused, the `play` method resumes playback at the point where it was paused. |
| `pause()` | Pauses the sound file. Subsequently calling the `play` method will cause the sound file to resume playing at the point where it was paused. |
| `stop()` | Stops the playback of the sound file. Subsequently calling the `play` method will cause the sound file to play from its beginning. |
| `setCycleCount(`*`value`*`)` | The argument is an `int` value that specifies the number of times that the sound file should be played. The default value is 1. If you pass the value `MediaPlayer.INDEFINITE`, the sound file will be repeated indefinitely. |

For example, the following code snippet loads the file *guitar.wav* and plays it as soon as the file is loaded and the `MediaPlayer` is ready:

```
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
MediaPlayer player = new MediaPlayer(media);
player.setAutoPlay(true);
```

**NOTE:** Neither the `Media` class nor the `MediaPlayer` class have any visual properties. Neither class is a subclass of `Node`, so you do not add objects of the `Media` class or the `MediaPlayer` class to your scene graph.

## Registering an `EndOfMedia` Event Handler

A `MediaPlayer` object has certain states that indicate what the `MediaPlayer` is currently doing. For example:

- When a sound file is loaded into memory and the `MediaPlayer` is ready to play it, the `MediaPlayer` is in the *READY* state.
- While the sound file is being played, the `MediaPlayer` is in the *PLAYING* state.
- While the sound file is paused, as a result of the `pause()` method, the `MediaPlayer` is in the *PAUSED* state.
- When the sound file has been stopped, as a result of the `stop()` method, the `MediaPlayer` is in the *STOPPED* state.

When a `MediaPlayer` object plays a sound file all the way to its end, the `MediaPlayer` object does not automatically transition to the *STOPPED* state. Instead, it remains in the *PLAYING* state. This can cause a problem in some applications, because the `play()` method will not play the sound file if the `MediaPlayer` object is already in the *PLAYING* state.

For example, suppose a program has a *play* button that should play a sound file every time the user clicks the button. The button's event handler calls the `MediaPlayer` object's `play()` method. The user clicks the *play* button and listens to the sound file all the way to its end. If the user clicks the *play* button again to replay the sound file, nothing happens because the `MediaPlayer` is already in the *PLAYING* state. In this type of application, you need a way to force the `MediaPlayer` object into the *STOPPED* state when it reaches the end of the sound file.

As it turns out, the `MediaPlayer` object fires an event when it reaches the end of the sound file. The event is known as the `EndOfMedia` event, and you can use the `setOnEndOfMedia` method to register an event handler that executes any time the event fires. In the event handler, you simply call the `MediaPlayer`'s `stop()` method to force the `MediaPlayer` into the *STOPPED* state. The following code snippet uses an anonymous inner class to create such an event handler:

```
// Load the sound file.
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
MediaPlayer player = new MediaPlayer(media);
```

```
// Register an OnEndOfMedia event handler.
player.setOnEndOfMedia(new Runnable()
{
   @Override
   public void run()
   {
      player.stop();
   }
});
```

The event handler for the `EndOfMedia` event must be a class that implements the `Runnable` interface. (The `Runnable` interface is in the `java.lang` package, so it does not require an `import` statement.) The `Runnable` interface is a functional interface that specifies one method, named `run()`. Because `Runnable` is a functional interface, we can simplify the event handling code by writing it as a lambda expression. Here is an example:

```
// Load the sound file.
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
MediaPlayer player = new MediaPlayer(media);

// Register an OnEndOfMedia event handler.
player.setOnEndOfMedia(() ->
{
      player.stop();
});
```

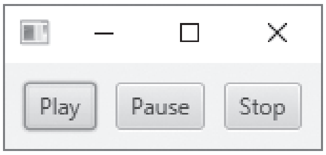
Code Listing 14-14 shows a complete example. The program loads the *guitar.wav* sound file, and it displays a play button, a pause button, and a stop button. These buttons allow the user to play, pause, and stop the sound file. The program's window is shown in Figure 14-36.

**Code Listing 14-14** (`SoundPlayer.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.control.Button;
import java.io.File;

public class SoundPlayer extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Load the sound file and create the MediaPlayer.
      File soundFile = new File("guitar.wav");
      Media media = new Media(soundFile.toURI().toString());
      MediaPlayer player = new MediaPlayer(media);

      // Event handler for the OnEndOfMedia event
      player.setOnEndOfMedia(() ->
      {
         player.stop();
      });

      // Create the buttons.
      Button playButton = new Button("Play");
      Button pauseButton = new Button("Pause");
      Button stopButton = new Button("Stop");

      // Event handler for the play Button
      playButton.setOnAction(event ->
      {
         player.play();
      });

      // Event handler for the pause Button
      pauseButton.setOnAction(event ->
      {
         player.pause();
      });

      // Event handler for the stop Button
      stopButton.setOnAction(event ->
      {
         player.stop();
      });

      // Put the Buttons in an HBox.
      HBox hbox = new HBox(10, playButton, pauseButton, stopButton);
      hbox.setAlignment(Pos.CENTER);
      hbox.setPadding(new Insets(10));

```

```
        // Create a Scene with the HBox as its root node.
        Scene scene = new Scene(hbox);

        // Add the Scene to the Stage and display the window.
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

**Figure 14-36** Window displayed by `SoundPlayer.java` (Oracle Corporate Counsel)



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

14.31 What two classes do you use to play an audio file?

14.32 Refer to your answer to Checkpoint 14.31. In what package are the two classes?

14.33 What does the `MediaPlayer` method `setAutoPlay` do?

14.34 What does the `MediaPlayer` method `setCycleCount` do?

## 14.5 Playing Videos

**CONCEPT: In JavaFX, you use the `Media`, `MediaPlayer`, and `MediaView` classes to play videos. The `Media` class loads a video file into memory, and the `MediaPlayer` class provides methods for playing the video. The `MediaView` class can be inserted into the scene graph, providing a way to see the video on the screen.**

VideoNote Playing Videos with JavaFX

Playing a video from a JavaFX application is similar to playing a sound file. First, you use the `Media` class to load the video (JavaFX supports the FLV and MP4 video formats), then you use the `MediaPlayer` class to control the video's playback. Because videos must be viewed on the screen, however, a third class named `MediaView` is required. (The `MediaView` class is in the `javafx.scene.media` package.) You simply create an instance of the `MediaView` class, passing the `MediaPlayer` object as an argument to the constructor.

The `MediaView` class is a subclass of `Node`, and you can add it to your scene graph. You set the size of the viewing window with the `setFitWidth` and `setFitHeight` methods. The video will be resized as necessary to fit within the viewing window. The program in Code Listing 14-15 shows an example. When you run this program, it loads a video named *TakeOff.mp4* and automatically plays it. An example of the program's output is shown in Figure 14-37.

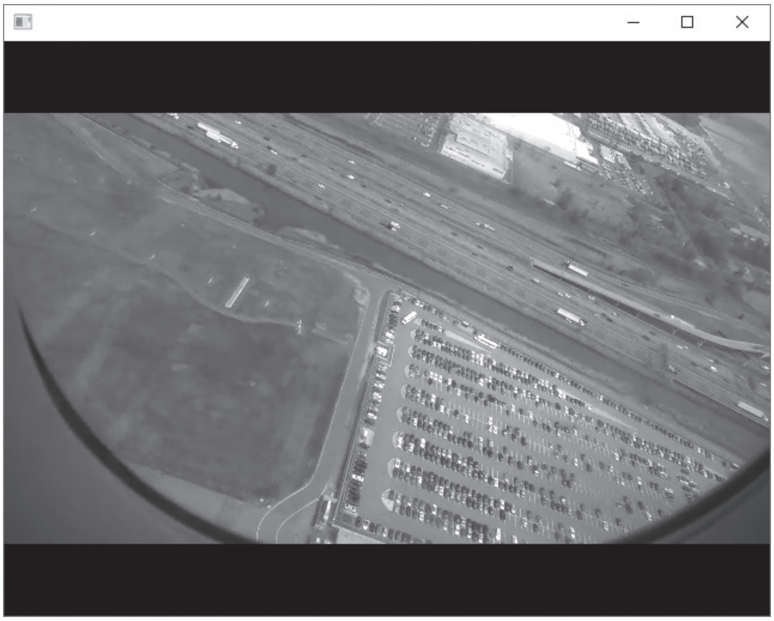
### **Code Listing 14-15** (VideoDemo.java)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import java.io.File;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;

public class VideoDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the video dimensions
      final double WIDTH = 640.0, HEIGHT = 480.0;

      // Load the video file.
      File videoFile = new File("TakeOff.mp4");
      Media media = new Media(videoFile.toURI().toString());

      // Create the MediaPlayer and set AutoPlay to true.
      MediaPlayer player = new MediaPlayer(media);
      player.setAutoPlay(true);

      // Create the MediaView.
      MediaView view = new MediaView(player);

      // Set the viewing dimensions.
      view.setFitWidth(WIDTH);
      view.setFitHeight(HEIGHT);

      // Put the MediaView in an HBox.
      HBox hbox = new HBox(view);

      // Create a Scene, put it on the stage, and display it.
      Scene scene = new Scene(hbox);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

Let’s take a closer look at the program:

- Lines 25 and 26 load a file named *TakeOff.mp4*, and create a `Media` object for the file.
- Line 29 creates a `MediaPlayer` object, and line 30 sets the object’s auto-play property to `true`. This causes the `MediaPlayer` object to automatically play the video once everything is set up.
- Line 33 creates a `MediaView` object.
- Lines 36 and 37 set the `MediaView` object’s width and height to 640 by 480.
- Line 40 puts the `MediaView` object in an `HBox`.
- Lines 43 through 45 create the scene graph with the `HBox` as its root node, sets the scene to the stage, and displays the application’s window.

**Figure 14-37** The `VideoDemo.java` application



In the section on *Playing Sound Files*, we discussed an issue with the `MediaPlayer` that sometimes has to be dealt with. When a `MediaPlayer` object plays a media file (sound or video) all the way to its end, the `MediaPlayer` object does not automatically transition to the *STOPPED* state. Instead, it remains in the *PLAYING* state. This can cause a problem in some applications because the `play()` method will not play the media file if the `MediaPlayer` object is already in the *PLAYING* state.

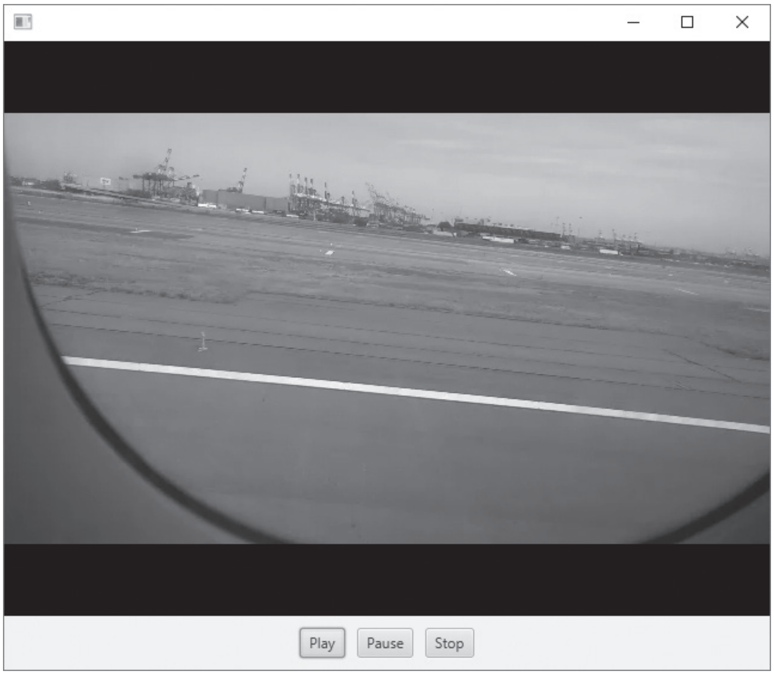
So, if you want the ability to replay a video by calling the `play()` method after the video has already played to its end, you have to force the `MediaPlayer` object into the *STOPPED* state. You can accomplish this by writing an event handler for the `EndOfMedia` event. In the event handler, you simply call the `MediaPlayer`’s `stop()` method.

Code Listing 14-16 shows a complete example. The program loads the *TakeOff.mp4* file, and it displays a *Play* button, a *Pause* button, and a *Stop* button. These buttons allow the user to play, pause, and stop the video. Notice in lines 36 through 39 we create an event handler for the `MediaPlayer` object's `EndOfMedia` event. Each time the video plays to its end, the event handler calls the `stop()` method, which forces the `MediaPlayer` into the *STOPPED* state. The program's window is shown in Figure 14-38.

**Code Listing 14-16** `(VideoPlayer.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import java.io.File;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.scene.control.Button;

public class VideoPlayer extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the video dimensions
      final double WIDTH = 640.0, HEIGHT = 480.0;

      // Load the video file.
      File videoFile = new File("TakeOff.mp4");
      Media media = new Media(videoFile.toURI().toString());

      // Create the MediaPlayer.
      MediaPlayer player = new MediaPlayer(media);

      // Event handler for the OnEndOfMedia event
      player.setOnEndOfMedia(() ->
      {
         player.stop();
      });

      // Create the MediaView.
      MediaView view = new MediaView(player);

      // Set the viewing dimensions.
      view.setFitWidth(WIDTH);
      view.setFitHeight(HEIGHT);

      // Create the buttons.
      Button playButton = new Button("Play");
      Button pauseButton = new Button("Pause");
      Button stopButton = new Button("Stop");

      // Event handler for the play Button
      playButton.setOnAction(event ->
      {
         player.play();
      });

      // Event handler for the pause Button
      pauseButton.setOnAction(event ->
      {
         player.pause();
      });

      // Event handler for the stop Button
      stopButton.setOnAction(event ->
      {
         player.stop();
      });

      // Put the Buttons in an HBox.
      HBox hbox = new HBox(10, playButton, pauseButton, stopButton);
      hbox.setAlignment(Pos.CENTER);
      hbox.setPadding(new Insets(10));

      // Put everything in a BorderPane.
      BorderPane borderPane = new BorderPane();
      borderPane.setCenter(view);
      borderPane.setBottom(hbox);

      // Create a Scene, put it on the stage, and display it.
      Scene scene = new Scene(borderPane);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 14-38** The `VideoPlayer.java` application



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

14.35 What three classes do you use to play a video file?

14.36 Refer to your answer to Checkpoint 14.35. In what package are the three classes?

14.37 How do you set the size of a video's viewing window?

## 14.6 Handling Key Events

**CONCEPT: JavaFX allows you to create event handlers that respond to events generated by the keyboard.**

VideoNote
Handling Key Events in JavaFX

*Key events* are events that are generated by the keyboard. There are three general types of key events:

- Any time the user presses a key, a `KEY_PRESSED` event occurs.
- Any time the user releases a key, a `KEY_RELEASED` event occurs.
- Any time the user presses and releases a key that produces a Unicode character, such as a letter of the alphabet, a numeric digit, or a punctuation symbol, a `KEY_TYPED` event occurs. (Keys such as the arrow keys, the page up and page down keys, and function keys do not generate `KEY_TYPED` events.)

When a key event occurs, an event object that is an instance of the `KeyEvent` class is created. `KeyEvent` is a subclass of the `Event` class, both of which are in the `javafx.event` package. Suppose we want to write an event handler class that can respond to key events.

This is the general format of an event handler class that handles event objects of the `KeyEvent` type:

```
class KeyHandler implements EventHandler<KeyEvent>
{
    @Override
    public void handle(KeyEvent event)
    {
        // Write event handling code here.
    }
}
```

Notice the first line of the class declaration ends with the clause `implements EventHandler<KeyEvent>`. This specifies that the class implements the `EventHandler` interface, and the type of event object that it will work with is `KeyEvent`. Next, notice the `handle` method has a `KeyEvent` parameter named `event`. When the `handle` method is called, a `KeyEvent` object will be passed to it as an argument. Any code you want to execute as the result of a key being pressed or released must be written in the `handle` method.

Once you have written the event handler class, the next step is to register it with the desired node. Any node in your scene graph can handle key events, as long as that node has the input focus. When a node has the *input focus*, it is ready to receive input. You can tell which node has the input focus, while a JavaFX application is running, because it usually has a blue "glow" around it. When you press the Tab key, the input focus moves from one node to the next.

In most cases, you will want to register your `KeyEvent` handlers with the `Scene` object. That way, your `KeyEvent` handlers will respond to key events, regardless of which node has the input focus. To register a key event handler, you will call one of the following methods on the `Scene` object (or any node in the scene graph):

- `setOnKeyPressed`. Use this method to register an event handler for `KEY_PRESSED` events.
- `setOnKeyReleased`. Use this method to register an event handler for `KEY_RELEASED` events.
- `setOnKeyTyped`. Use this method to register an event handler for `KEY_TYPED` events.

## Using an Anonymous Inner Class to Register a Key Event Handler to the Scene

You learned in Chapter 12 that a common technique for writing event handlers is to use anonymous inner classes. The following code snippet shows what it looks like to use an anonymous inner class to register a *KEY_PRESSED* event handler for the scene:

```
// Create a Scene.
Scene scene = new Scene(rootNode);

// Register a KEY_PRESSED handler for the scene.
scene.setOnKeyPressed(new EventHandler<KeyEvent>()
{
        @Override
        public void handle(KeyEvent event)
        {
            // Write event-handling code here.
        }
    });
```

## Using a Lambda Expression to Register a Key Event Handler to the Scene

You also learned in Chapter 12 that lambda expressions provide a simplified way of registering event handlers. The following code snippet shows what it looks like to use a lambda expression to register a *KEY_PRESSED* event handler for the scene:

```
// Create a Scene.
Scene scene = new Scene(rootNode);

// Register a KEY_PRESSED handler for the scene.
scene.setOnKeyPressed(event ->
{
    // Write event-handling code here.
});
```

Code Listing 14-17 shows a complete program that demonstrates how to handle *KEY_PRESSED* events. The application simply increments a counter each time a key is pressed, and displays the value of the counter in a `Label` control. The application's window, after five key presses, is shown in Figure 14-39.

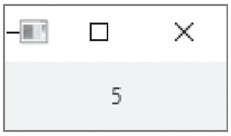
**Code Listing 14-17** (`KeyPressedDemo.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.geometry.Pos;
import javafx.geometry.Insets;
import javafx.scene.control.Label;
import javafx.event.EventHandler;
import javafx.scene.input.KeyEvent;

public class KeyPressDemo extends Application
{
   private int count = 0;     // To keep count of key presses

   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }
20
21     @Override
22     public void start(Stage primaryStage)
23     {
24        // Create a Label control.
25        Label label = new Label("0");
26
27        // Put the Label in an HBox.
28        HBox hbox = new HBox(10, label);
29        hbox.setAlignment(Pos.CENTER);
30        hbox.setPadding(new Insets(10));
31
32        // Create a Scene with the HBox as its root node.
33        Scene scene = new Scene(hbox);
34
35        // Register a KEY_PRESSED handler for the scene.
36        scene.setOnKeyPressed(event ->
37        {
38           count++;
39           label.setText(String.format(”%d”, count));
40        });
41
42        // Set the scene on the stage and dispaly it.
43        primaryStage.setScene(scene);
44        primaryStage.show();
45     }
46  }
```

Let's take a closer look at the program:

- Line 13 declares an `int` field named `count`, initialized with the value 0. The `count` field will keep count of the number of key presses.
- Line 25 creates a `Label` control that will display the number of key presses, and line 28 puts the `Label` control in an `HBox`.
- Line 33 creates a scene, with the `HBox` as the root node.
- Lines 36 through 40 use a lambda expression to register an event handler with the `Scene` object. The event handler will respond to `KEY_PRESSED` events. Inside the event handler, line 38 increments `count`, and line 39 displays the value of `count` in the `Label` control.
- Lines 43 and 44 set the scene to the stage and display the application's window.

**Figure 14-39** The `KeyPressedDemo.java` application after 5 keys have been pressed (Oracle Corporate Counsel)



5

The `KeyEvent` object that is passed as an argument to the event handler contains data about the key event. For example, you can use the object's `getCode()` method to determine which key was pressed to generate the event. The `getCode()` method returns a `KeyCode` value, which is an `enum` in the `javafx.scene.input` package. The `KeyCode` enum contains an extensive set of constants representing the keys on a computer keyboard. For example, `KeyCode.J` represents the J key, `KeyCode.ENTER` represents the Enter key, and `KeyCode.PAGE_UP` represents the Page Up key. See the JavaFX API documentation for the `KeyCode` enumeration for a list of all the constants.

Code Listing 14-18 shows an example that lets the user move a circle around the screen using the arrow keys. An example of the program's screen is shown in Figure 14-40.

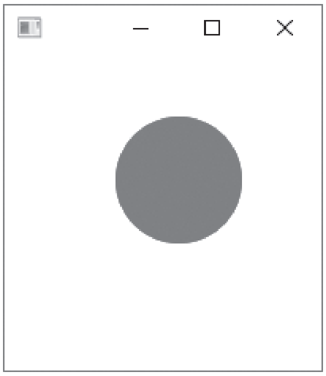
**Code Listing 14-18** (`MoveBall.java`)

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
import javafx.scene.layout.Pane;
import javafx.event.EventHandler;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.KeyCode;

public class MoveBall extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 200.0;
      final double SCENE_HEIGHT = 200.0;

      // Constants for the ball
      final double START_X = 100.0, START_Y = 40.0;
      final double RADIUS = 40.0;
      final double STEP = 10.0;

      // Create the ball.
      Circle ball = new Circle(START_X, START_Y, RADIUS);
      ball.setFill(Color.RED);

      // Add the ball to a Pane.
      Pane pane = new Pane(ball);

      // Create a Scene and display it.
      Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);

      // Register a KEY_PRESSED handler for the scene.
      scene.setOnKeyPressed(event ->
      {
         // Get the ball's coordinates.
         double x = ball.getCenterX();
         double y = ball.getCenterY();

         // Check for the down-arrow key.
         if (event.getCode() == KeyCode.DOWN &&
             y < (SCENE_HEIGHT - RADIUS))
         {
            ball.setCenterY(y + STEP);
         }

         // Check for the up-arrow key.
         if (event.getCode() == KeyCode.UP &&
             y > RADIUS)
         {
            ball.setCenterY(y - STEP);
         }

         // Check for the right-arrow key.
         if (event.getCode() == KeyCode.RIGHT &&
             x < (SCENE_WIDTH - RADIUS))
         {
            ball.setCenterX(x + STEP);
         }

         // Check for the left-arrow key.
         if (event.getCode() == KeyCode.LEFT &&
             x > RADIUS)
         {
            ball.setCenterX(x - STEP);
         }
      });

      // Set the scene on the stage and display it.
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

Let's take a closer look at the program:

- Lines 22 and 23 define constants for the scene dimensions.
- Lines 26 through 28 define constants for the `Circle` object.
- Line 31 creates the `Circle` object, and line 32 sets its color to red. Line 35 adds the `Circle` object to a `Pane`, and line 38 creates a scene, with the `Pane` as the root node.
- Lines 41 through 74 uses a lambda expression to register an event handler for the scene. The event handler will respond to *KEY_PRESSED* events.
    - Lines 44 and 45 get the coordinates of the `Circle` object's center point, and assign them to the variables `x` and `y`.
    - The `if` statement in lines 48 through 52 determines whether the key that was pressed was the down-arrow, and whether the bottom of the `Circle` object is above the bottom edge of the window. If these conditions are true, line 51 moves the `Circle` object down by 10 pixels.
    - The `if` statement in lines 55 through 59 determines whether the key that was pressed was the up-arrow, *and* whether the top of the `Circle` object is below the top edge of the window. If these conditions are true, line 58 moves the `Circle` object up by 10 pixels.
    - The `if` statement in lines 62 through 66 determines whether the key that was pressed was the right-arrow, *and* whether the right side of the `Circle` object is not at the right edge of the window. If these conditions are true, line 65 moves the `Circle` object to the right by 10 pixels.
    - The `if` statement in lines 69 through 73 determines whether the key that was pressed was the left-arrow, *and* whether the left side of the `Circle` object is not at the left edge of the window. If these conditions are true, line 72 moves the `Circle` object to the left by 10 pixels.
- Lines 77 and 78 set the scene to the stage and display the application's window.

**Figure 14-40** The `MoveBall.java` application (Oracle Corporate Counsel)



**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

14.38 What type of event happens when the user presses a key on the keyboard?

14.39 What type of event happens when the user releases a key on the keyboard?

14.40 What type of event happens when the user presses and releases a key that produces a Unicode character?

14.41 What `KeyEvent` method can you call to determine which key the user pressed?

14.42 Refer to your answer for Checkpoint 14.41. What does the method return?

## 14.7 Handling Mouse Events

**CONCEPT: JavaFX allows you to create event handlers that respond to events generated by the mouse.**

VideoNote
Handling Mouse Events in JavaFX

The mouse generates several types of events, most of which are described in Table 14-25. When a mouse event occurs, a `MouseEvent` object is created and delivered to the node that is under the mouse cursor. For example, if the user clicks the mouse button while the mouse cursor is over an `ImageView` control, a `MouseEvent` object will be delivered to the `ImageView` control. Likewise, if the user drags the mouse while the mouse cursor is over a `Circle` node, a `MouseEvent` object will be delivered to the `Circle` node.

**Table 14-25** Mouse Event Types

| Mouse Event Type | Description |
|---|---|
| `MOUSE_CLICKED` | This event occurs when the user presses and releases the mouse button. |
| `MOUSE_DRAGGED` | This event occurs when the user moves the mouse while pressing the mouse button. |
| `MOUSE_ENTERED` | This event occurs when the mouse cursor enters the space occupied by a node. |
| `MOUSE_EXITED` | This event occurs when the mouse cursor exits the space occupied by a node. |
| `MOUSE_MOVED` | This event occurs when the mouse is moved without the mouse button being pressed. |
| `MOUSE_PRESSED` | This event occurs when the user presses the mouse button. |
| `MOUSE_RELEASED` | This event occurs when the user releases the mouse button. |

It is the responsibility of each node to handle (or ignore) any mouse events that are delivered to the node. For example, if your application displays a `Rectangle`, and you want an action to take place when the user clicks the `Rectangle`, then you must write an event handler that responds to `MOUSE_CLICKED` events, and register that event handler with the `Rectangle`.

To register a mouse event handler, you will call one of the following methods on the node that should handle the event:

- `setOnMouseClicked`. Use this method to register an event handler for `MOUSE_CLICKED` events.
- `setOnMouseDragged`. Use this method to register an event handler for `MOUSE_DRAGGED` events.
- `setOnMouseEntered`. Use this method to register an event handler for `MOUSE_ENTERED` events.
- `setOnMouseExited`. Use this method to register an event handler for `MOUSE_EXITED` events.

- `setOnMouseMoved`. Use this method to register an event handler for `MOUSE_MOVED` events.
- `setOnMousePressed`. Use this method to register an event handler for `MOUSE_PRESSED` events.
- `setOnMouseReleased`. Use this method to register an event handler for `MOUSE_RELEASED` events.

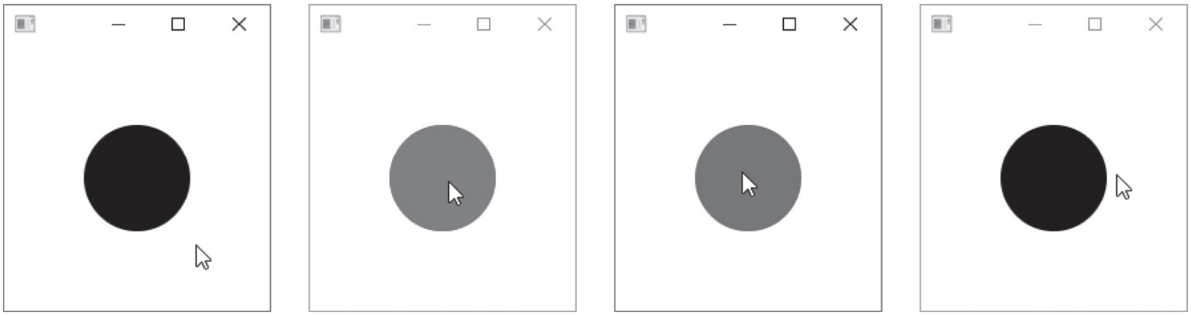
Code Listing 14-19 shows a complete program that demonstrates how to handle mouse events. The program draws a circle, filled with the default color of black, in the center of the window. Then, in lines 31 through 44, it registers event handlers for the `MOUSE_ENTERED`, `MOUSE_EXITED`, and `MOUSE_PRESSED` events. When the mouse cursor moves into the circle, the circle turns red. When the mouse cursor moves out of the circle, the circle turns back to black. When the user clicks the circle, the circle turns green. This is described in Figure 14-41.

**Code Listing 14-19** (`MouseEventDemo.java`)

```
 1  import javafx.application.Application;
 2  import javafx.stage.Stage;
 3  import javafx.scene.Scene;
 4  import javafx.scene.shape.Circle;
 5  import javafx.scene.paint.Color;
 6  import javafx.scene.layout.Pane;
 7
 8  public class MouseEventDemo extends Application
 9  {
10     public static void main(String[] args)
11     {
12        // Launch the application.
13        launch(args);
14     }
15
16     @Override
17     public void start(Stage primaryStage)
18     {
19        // Constants for the scene size
20        final double SCENE_WIDTH = 200.0;
21        final double SCENE_HEIGHT = 200.0;
22
23        // Constants for the ball
24        final double CENTER_X = 100.0, CENTER_Y = 100.0;
25        final double RADIUS = 40.0;
26
27        // Create the ball.
28        Circle ball = new Circle(CENTER_X, CENTER_Y, RADIUS);
29
```

```
      // Register mouse event handlers with the ball.
      ball.setOnMouseEntered(event ->
      {
         ball.setFill(Color.RED);
      });

      ball.setOnMouseExited(event ->
      {
         ball.setFill(Color.BLACK);
      });

      ball.setOnMousePressed(event ->
      {
         ball.setFill(Color.GREEN);
      });

      // Add the ball to a Pane.
      Pane pane = new Pane(ball);

      // Create a Scene and display it.
      Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

**Figure 14-41** The `MouseEventDemo.java` application (Oracle Corporate Counsel)



The circle is initially filled with the default color, black.

When the mouse cursor enters the circle, the circle turns red.

When the user clicks the circle, the circle turns green.

When the mouse cursor exits the circle, the circle turns black.

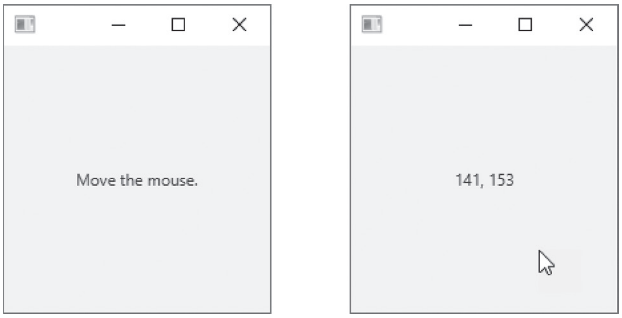
In your mouse event handler, if you want to get the mouse cursor's location, you can call the `MouseEvent` object's `getSceneX` and `getSceneY` methods. These methods return the mouse cursor's *X* and *Y* coordinates, respectively. For example, look at Code Listing 14-20. The program displays a window with a `Label` control that tells the user to "Move the mouse." When the user moves the mouse inside the application's window, the (*X*, *Y*) coordinates of the mouse cursor are continuously displayed in the `Label` control. The program's output is shown in Figure 14-42.

**Code Listing 14-20** (MouseMovedDemo.java)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;
import javafx.geometry.Pos;

public class MouseMovedDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 200.0;
      final double SCENE_HEIGHT = 200.0;

      // Create a Label.
      Label label = new Label("Move the mouse.");

      // Add the Label to a Pane.
      HBox hbox = new HBox(label);
      hbox.setAlignment(Pos.CENTER);

      // Create a Scene.
      Scene scene = new Scene(hbox, SCENE_WIDTH, SCENE_HEIGHT);

      // Register an event handler to the scene for
      // the MOUSE_MOVED event.
      scene.setOnMouseMoved(event ->
      {
         // Get the mouse cursor's coordinates.
         double x = event.getSceneX();
         double y = event.getSceneY();

         // Display the mouse cursor's location.
         label.setText(String.format("%.0f, %.0f", x, y));
      });

45          // Set the scene to the stage and display it.
46          primaryStage.setScene(scene);
47          primaryStage.show();
48       }
49    }
```

Notice in Code Listing 14-20 the mouse event handler (in lines 35 through 43) is registered to the `Scene` object, not the `Label` control. This is because we want to get mouse events as long as the mouse cursor is anywhere inside the application's window. If we had registered the event handler to the `Label` control, we would get mouse events only when the mouse cursor was over the `Label` control

**Figure 14-42** The `MouseMovedDemo.java` application (Oracle Corporate Counsel)



Move the mouse.

141, 153

The initial message.

The mouse cursor's coordinates displayed.

The `MOUSE_DRAGGED` event fires when the user moves the mouse while pressing the mouse button. The event continues to fire while the user performs this action. Code Listing 14-21 demonstrates this. The program draws a circle, initially positioned in the center of the window. It then registers an event handler that lets the user drag the circle with the mouse.

**Code Listing 14-21** (`MouseDraggedDemo.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;

public class MouseDraggedDemo extends Application
{
   public static void main(String[] args)
   {
      // Launch the application.
      launch(args);
   }

15     @Override
16     public void start(Stage primaryStage)
17     {
18        // Constants for the scene size
19        final double SCENE_WIDTH = 200.0;
20        final double SCENE_HEIGHT = 200.0;
21 
22        // Constants for the ball
23        final double CENTER_X = 100.0, CENTER_Y = 100.0;
24        final double RADIUS = 40.0;
25 
26        // Create the ball.
27        Circle circle = new Circle(CENTER_X, CENTER_Y, RADIUS);
28 
29        // Register an event handler for
30        // the MOUSE_DRAGGED event.
31        circle.setOnMouseDragged(event ->
32        {
33           // Get the mouse cursor's coordinates.
34           double x = event.getSceneX();
35           double y = event.getSceneY();
36 
37           // Move the circle.
38           circle.setCenterX(x);
39           circle.setCenterY(y);
40        });
41 
42        // Add the circle to a Pane.
43        Pane pane = new Pane(circle);
44 
45        // Create a Scene and display it.
46        Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
47        primaryStage.setScene(scene);
48        primaryStage.show();
49     }
50  }
```

Let's take a closer look at the mouse event handler, which is registered to the `Circle` object in lines 31 through 40. Lines 34 and 35 get the mouse cursor's coordinates and assigns them to the `x` and `y` variables. Then, lines 38 and 39 move the circle's center point to these coordinates. As a result, the circle's center point is moved to the same location as the mouse cursor.

## 14.8 Common Errors to Avoid

- **Trying to use Cartesian coordinates when drawing shapes.** The screen coordinate system differs from the Cartesian coordinate system that you learned about in mathematics. In the Cartesian coordinate system, the *Y* coordinates decrease as you move downward. In the screen coordinate system, the *Y* coordinates increase as you move downward, toward the bottom of the screen.
- **Specifying too many vertices for the `Polygon` class.** The `Polygon` class draws a closed polygon, so it automatically draws a line segment connecting the ending vertex with the first vertex. It is a common mistake to specify the same values for both the starting vertex and the ending vertex.
- **Rotating a node in the wrong direction with a `RotationTransition`.** If you pass a positive angle to the `setToAngle` or the `setByAngle` methods, the rotation will be clockwise. If you pass a negative angle to either of these methods, the rotation will be counterclockwise.
- **Trying to combine effects using the `Node` class's `setEffect` method.** The `Node` class's `setEffect` method accepts only one effect object as an argument. If you call the `setEffect` method more than once for a particular node, the last effect object that you pass as an argument replaces any previously passed effect objects for that node. However, it is possible to combine effects using the `Effect` class's `setInput` method.
- **Registering a `KeyEvent` handler with the wrong node.** In most cases, you will want to register your `KeyEvent` handlers with the `Scene` object. That way, your `KeyEvent` handlers will respond to key events, regardless of which node has the input focus.

## Review Questions and Exercises

### Multiple Choice and True/False

1. `Line`, `Circle`, and `Rectangle` are subclasses of the __________ class.
   a. `Sprite`
   b. `Geometry`
   c. `BasicShape`
   d. `Shape`
2. Assume `myCircle` is a `Circle` object. Which of these statements causes `myCircle` to not be filled with a color?
   a. `myCircle.setColor(null);`
   b. `myCircle.setStroke(null);`
   c. `myCircle.setFill(null);`
   d. `myCircle.setFill(Color.NONE);`
3. The __________ arc type causes a straight line to be drawn from one endpoint of the arc to the other endpoint.
   a. `ArcType.CHORD`
   b. `ArcType.ROUND`
   c. `ArcType.OPEN`
   d. `ArcType.CLOSED`

4. The __________ arc type causes straight lines to be drawn from each endpoint to the arc's center point. As a result, the arc will be shaped like a pie slice.
   a. `ArcType.CHORD`
   b. `ArcType.ROUND`
   c. `ArcType.OPEN`
   d. `ArcType.CLOSED`
5. The __________ arc type causes no lines to connect the endpoints. Only the arc will be drawn.
   a. `ArcType.CHORD`
   b. `ArcType.ROUND`
   c. `ArcType.OPEN`
   d. `ArcType.CLOSED`
6. Which `Node` class method do you call to rotate a node about its center?
   a. `rotate()`
   b. `rotateCenter()`
   c. `setRotate()`
   d. `centerRotate()`
7. This class creates an animation in which a node moves from one position to another over a period of time.
   a. `TranslateTransition`
   b. `MoveTransition`
   c. `MotionTransition`
   d. `PositionTransition`
8. You use this class to specify the amount of time that an animation should last.
   a. `TimeSpan`
   b. `Duration`
   c. `Clock`
   d. `AnimationTime`
9. This interpolator causes an animation to start slowly, accelerate, then slow down at the end. This is the default interpolator.
   a. `Interpolator.SLOW`
   b. `Interpolator.EASE_START_END`
   c. `Interpolator.GRADUAL`
   d. `Interpolator.EASE_BOTH`
10. You use these two classes to play an audio file.
    a. `Media` and `MediaPlayer`
    b. `Audio` and `AudioPlayer`
    c. `Sound` and `SoundPlayer`
    d. `LoadMedia` and `PlayMedia`
11. **True or False:** In a 640 by 480 window, the coordinates of the pixel in the upper-left corner are (0, 0).
12. **True or False:** In a 640 by 480 window, the coordinates of the pixel in the lower-right corner are (640, 480).
13. **True or False:** If an ellipse's *X*-radius and *Y*-radius have the same value, then the ellipse will be a circle.

14. **True or False:** The `Node` class's `setEffect` method accepts multiple effect objects as arguments, allowing you to combine effects.
15. **True or False:** Any time the user releases a key, a `KEY_PRESSED` event occurs.

## Algorithm Workbench

1. Write a statement that instantiates the `Line` class, and uses the constructor to draw a line from (50, 25) to (150, 125).
2. Write a statement that instantiates the `Circle` class, and uses the constructor to draw a circle with its center point at (100, 75) and with a radius of 120.
3. Write a statement that instantiates the `Rectangle` class, and uses the constructor to draw a rectangle with its upper-left corner at (100, 200), with a width of 175, and a height of 150.
4. Write code that does the following: Instantiates the `Ellipse` class, and uses the constructor to draw an ellipse with its center located at (300, 200), an x-radius of 120 pixels, and a y-radius of 100. Then, set the ellipse's stroke color to black, and the fill color to red.
5. Write a statement that instantiates the `Polygon` class, and uses the constructor to draw a polygon with vertices at the following points: (100, 100), (200, 100), (200, 200), and (100, 200).
6. Write a statement that instantiates the `Text` class, and uses the constructor to draw the string "Hello World", starting at the coordinates 100, 200.
7. Assume `myBox` is a `Rectangle` object. Write the statement that rotates `myBox` 45 degrees about its center.
8. Assume `myBox` is a `Rectangle` object. Write the code that scales `myBox` to half its current size, along both the *X* and *Y* axes.
9. Write code that creates a `Circle`, with a radius of 30, and then creates a `TranslateTransition` object to animate it. The duration of the animation is 5 seconds. When the animation is played, the `Circle` object will start at (0, 100), and it will move to the location (200, 150). It will take 3 seconds for the `Circle` object to complete the move.
10. Write code that loads the audio file `C:\media\saxaphone.wav`, and creates a `MediaPlayer` object that is ready to play the file.
11. Write code that loads the video file `meteor_shower.mp4`, and creates a `MediaViewer` object that is ready to play the file in a viewing window that is 640 by 480 in size.

## Short Answer

1. In a 640 by 480 window, what are the coordinates of the pixel in the upper-right corner of the window?
2. How would you use the `Ellipse` class to draw a circle?
3. What `TranslateTransition` class methods do you use to specify a node's starting location?
4. What `TranslateTransition` class methods do you use to specify a node's ending location?
5. What `RotateTransition` class method do you use to specify a node's starting angle?
6. What `RotateTransition` class method do you use to specify a node's ending angle?

7. What `RotateTransition` class method do you use to specify the amount of rotation?
8. What `ScaleTransition` class methods do you use to specify a node's starting scale factor?
9. What `ScaleTransition` class methods do you use to specify a node's ending scale factor?
10. What two effect classes can you use to create shadow effects?
11. What three effect classes can you use to create blur effects?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. This Old House

Use the basic shapes you learned in this chapter to draw a house. Be sure to include at least two windows and a door. Feel free to draw other objects as well, such as the sky, sun, and even clouds.

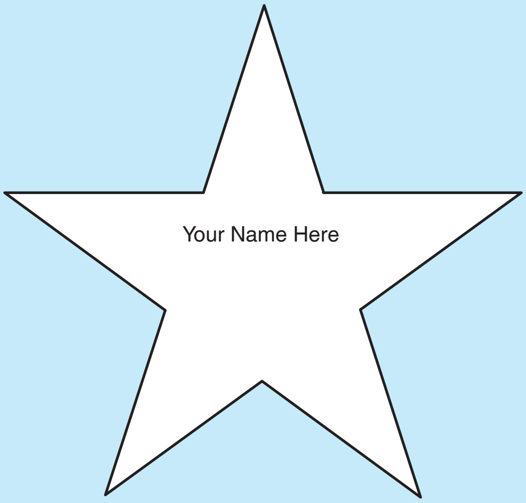
### 2. Tree Age

Counting the growth rings of a tree is a good way to tell the age of a tree. Each growth ring counts as one year. Use either the `Circle` or `Ellipse` classes to draw how the growth rings of a 5-year-old tree might look. Then, using the `Text` shape class, number each growth ring starting from the center and working outward with the age in years associated with that ring.

### 3. Hollywood Star

Make your own star on the Hollywood Walk of Fame. Write a program that displays a star similar to the one shown in Figure 14-43, with your name displayed in the star.

**Figure 14-43** Hollywood star



### 4. Vehicle Outline

Using the shapes you learned about in this chapter, draw the outline of the vehicle of your choice (car, truck, airplane, and so forth).

### 5. Solar System

Use the `Circle` class to draw each of the planets of our solar system. Draw the sun first, then each planet according to distance from the sun. (Mercury, Venus, Earth, Mars, Jupiter Saturn, Uranus, Neptune, and the dwarf planet, Pluto.) Label each planet using the `Text` shape class.

### 6. Opacity Adjuster

Create an application that lets the user select an image file via a `FileChooser` dialog. The application should display the image, and provide a `Slider` control. The `Slider` control should let the user adjust the image's opacity.

### 7. Color Adjuster

Create an application that lets the user select an image file via a `FileChooser` dialog. The application should display the image and provide four `Slider` controls. The `Slider` controls should let the user adjust the image's hue, saturation, brightness, and contrast.

### 8. Mouse Rollover

Create an application that displays an image. When the user moves the mouse pointer over the image, it should change to a second image. The second image should remain displayed as long as the mouse pointer is over it. When the user moves the mouse pointer away from the second image, it should change back to the first image.

### 9. Coin Toss

VideoNote
The Coin Toss Problem

Write a program that simulates the tossing of a coin. The program should wait for the user to press a key, then generate a random number in the range of 0 through 1. If the number is 0, the program should display the image of a coin with heads facing up. If the number is 1, the program should display the image of a coin with tails facing up. (You can create your own coin images, or use the ones provided in the book's online resources, downloadable from www.pearsonhighered.com/gaddis.)

### 10. Lunar Lander

The book's online resources (downloadable from www.pearsonhighered.com/gaddis) provide images of a spacecraft and a background drawing of the moon's surface. Write a program that initially displays the spacecraft on the moon's surface. When the user presses the spacebar, the spacecraft should slowly lift off the surface.

### 11. Change for a Dollar Game

The book's online resources (downloadable from www.pearsonhighered.com/gaddis) provide images of a penny, a nickel, a dime, and a quarter. Create a game that displays each of these images, plus the text "Count Change". The game should let the user click any of the coins, in any order. To win the game, the user must click the coins that, when added together, equal exactly one dollar. When the user clicks the "Count Change" text, the program should show the amount of money that the user clicked. If the amount equals one dollar, the program should indicate that the user won the game.

### 12. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows:

1. When the program begins, a random number in the range of 0 through 2 is generated. If the number is 0, then the computer has chosen rock. If the number is 1, then the computer has chosen paper. If the number is 2, then the computer has chosen scissors. (Don't display the computer's choice yet.)
2. To make his or her selection, the user clicks an image on the screen. (You can find images for this game included in the book's online resources at www.pearsonhighered.com/gaddis.)
3. The computer's choice is displayed.
4. A winner is selected according to the following rules:
   - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
   - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
   - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
   - If both players make the same choice, the game is a tie.

# CHAPTER 15 Recursion

## TOPICS

15.1 Introduction to Recursion
15.2 Solving Problems with Recursion
15.3 Examples of Recursive Methods
15.4 A Recursive Binary Search Method
15.5 The Towers of Hanoi
15.6 Common Errors to Avoid

## 15.1 Introduction to Recursion

**CONCEPT: A recursive method is a method that calls itself.**

You have seen instances of methods calling other methods. Method A can call method B, which can then call method C. It's also possible for a method to call itself. A method that calls itself is a recursive method. Look at the `message` method in Code Listing 15-1.

**Code Listing 15-1** (`EndlessRecursion.java`)

```
/**
   This class has a recursive method.
*/

public class EndlessRecursion
{
   public static void message()
   {
      System.out.println("This is a recursive method.");
      message();
   }
}
```

This method displays the string `"This is a recursive method."` and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the method? There's no way to stop the recursive calls. This method is like an infinite loop because there is no code to stop it from repeating.

Like a loop, a recursive method must have some way to control the number of times it repeats. The class in Code Listing 15-2 has a modified version of the `message` method. It passes an integer argument, which holds the number of times the method should call itself.

**Code Listing 15-2** `(Recursive.java)`

```
/**
   This class has a recursive method, message,
   which displays a message n times.
*/

public class Recursive
{
   public static void message(int n)
   {
      if (n > 0)
      {
         System.out.println("This is a recursive method.");
         message(n - 1);
      }
   }
}
```

This method contains an `if` statement that controls the repetition. As long as the `n` parameter is greater than zero, the method displays the message and calls itself again. Each time it calls itself, it passes `n - 1` as the argument. For example, look at the program in Code Listing 15-3.

**Code Listing 15-3** `(RecursionDemo.java)`

```
/**
   This class demonstrates the Recursive.message method.
*/

public class RecursionDemo
{
   public static void main(String[] args)
   {
      Recursive.message(5);
      }
   }

```

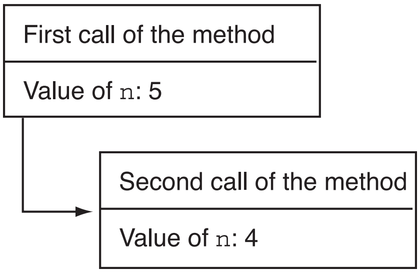
**Program Output**

```
This is a recursive method.
This is a recursive method.
This is a recursive method.
This is a recursive method.
This is a recursive method.
```

The `main` method in this class calls the `Recursive.message` method with the argument 5, which causes the method to call itself five times. The first time the method is called, the `if` statement displays the message and then calls itself with 4 as the argument. Figure 15-1 illustrates this.

The diagram in Figure 15-1 illustrates two separate calls of the `message` method. Each time the method is called, a new instance of the `n` parameter is created in memory. The first time the method is called, the `n` parameter is set to 5. When the method calls itself, a new instance of `n` is created, and the value 4 is passed into it. This cycle repeats until finally, zero is passed to the method. This is illustrated in Figure 15-2.

**Figure 15-1** First two calls of the method



As you can see from Figure 15-2, the method is called a total of six times. The first time it is called from the `main` method of the `RecursionDemo` class, and the other five times it calls itself. The number of times that a method calls itself is known as the *depth of recursion*. In this example, the depth of recursion is five. When the method reaches its sixth call, the `n` parameter is set to 0. At that point, the `if` statement's conditional expression is `false`, so the method returns. Control of the program returns from the sixth instance of the method to the point in the fifth instance directly after the recursive method call. This is illustrated in Figure 15-3.

Because there are no more statements to be executed after the method call, the fifth instance of the method returns control of the program back to the fourth instance. This repeats until all instances of the method return.

**Figure 15-2** Total of six calls to the `message` method

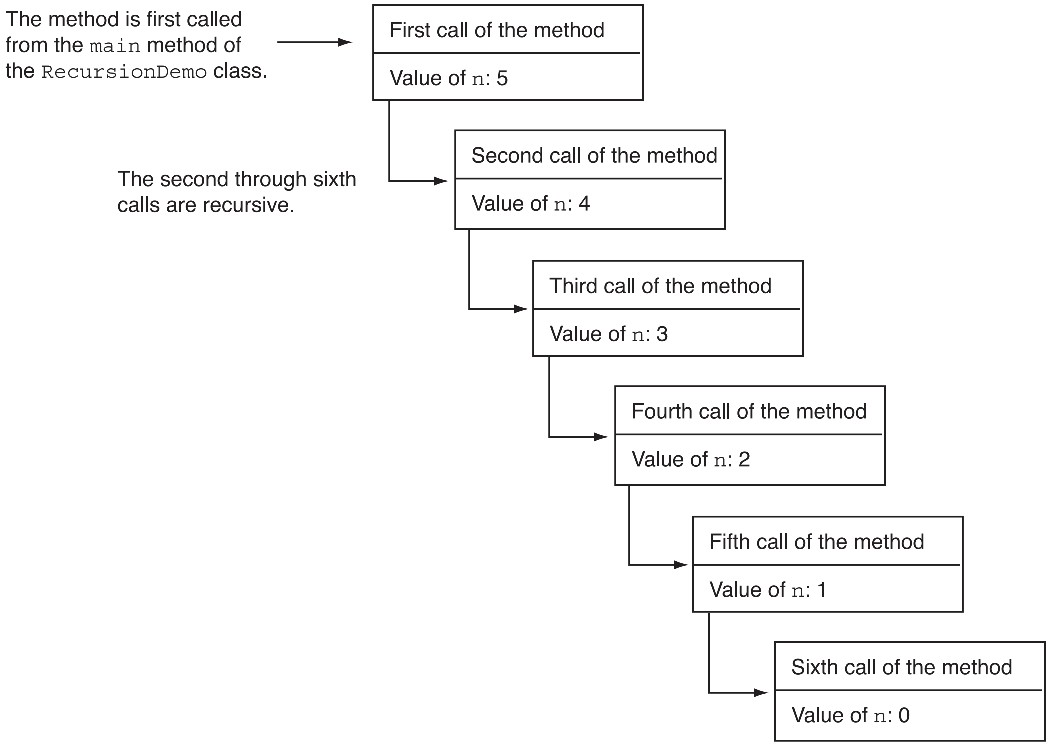

The method is first called from the main method of the RecursionDemo class.
The second through sixth calls are recursive.
First call of the method
Value of n: 5
Second call of the method
Value of n: 4
Third call of the method
Value of n: 3
Fourth call of the method
Value of n: 2
Fifth call of the method
Value of n: 1
Sixth call of the method
Value of n: 0

**Figure 15-3** Control returns to the point after the recursive method call

```
public static void message(int n)
{
     if (n > 0)
     {
          System.out.println("This is a recursive method.");
          message(n - 1);
     }
}
```

Recursive method call

Control returns here from the recursive call. There are no more statements to execute in this method, so the method returns.

## 15.2 Solving Problems with Recursion

**CONCEPT: A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.**

The `Recursive` and `RecursionDemo` classes shown in the previous section demonstrate the mechanics of a recursive method. Recursion can be a powerful tool for solving repetitive problems and is an important topic in upper level computer science courses. What might not be clear to you yet is how to use recursion to solve a problem. First, it should be noted that recursion is never absolutely required to solve a problem. Any problem that can be solved recursively can also be solved iteratively, with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because a method call requires several actions to be performed by the JVM. These actions include allocating memory for parameters and local variables, and storing the address of the program location where control returns after the method terminates. These actions, which are sometimes referred to as overhead, take place with each method call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with iteration. Where an iterative algorithm might result in faster execution time, the programmer might be able to design a recursive algorithm faster.

In general, a recursive method works like this:

VideoNote
Reducing a Problem with Recursion

- If the problem can be solved now, without recursion, then the method solves it and returns.
- If the problem cannot be solved now, then the method reduces it to a smaller but similar problem and calls itself to solve the smaller problem.

In order to apply this approach, first we identify at least one case in which the problem can be solved without recursion. This is known as the *base case*. Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case*. In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.

Let's take an example from mathematics to examine an application of recursion. In mathematics, the notation $n!$ represents the factorial of the number $n$. The factorial of a nonnegative number can be defined by the following rules:

If $n = 0$ then $n! = 1$
If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n$

Let's replace the notation $n!$ with factorial($n$), which looks a bit more like computer code, and rewrite these rules as follows:

If $n = 0$ then $\text{factorial}(n) = 1$
If $n > 0$ then $\text{factorial}(n) = 1 \times 2 \times 3 \times \ldots \times n$

These rules state that when $n$ is 0, its factorial is 1. When $n$ is greater than 0, its factorial is the product of all the positive integers from 1 up to $n$. For instance, factorial(6) is calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$.

When designing a recursive algorithm to calculate the factorial of any number, first we identify the base case, which is the part of the calculation that we can solve without recursion. That is the case where $n$ is equal to 0 as follows:

If $n = 0$ then $\text{factorial}(n) = 1$

This tells how to solve the problem when $n$ is equal to 0, but what do we do when $n$ is greater than 0? That is the recursive case, or the part of the problem that we use recursion to solve. This is how we express it:

If $n > 0$ then $\text{factorial}(n) = n \times \text{factorial}(n - 1)$

This states that if $n$ is greater than 0, the factorial of $n$ is $n$ times the factorial of $n - 1$. Notice how the recursive call works on a reduced version of the problem, $n - 1$. So, our recursive rule for calculating the factorial of a number might look like this:

| | |
|---|---|
| If $n = 0$ then | $\text{factorial}(n) = 1$ |
| If $n > 0$ then | $\text{factorial}(n) = n \times \text{factorial}\ (n - 1)$ |

The following code shows how this might be implemented in a Java method:

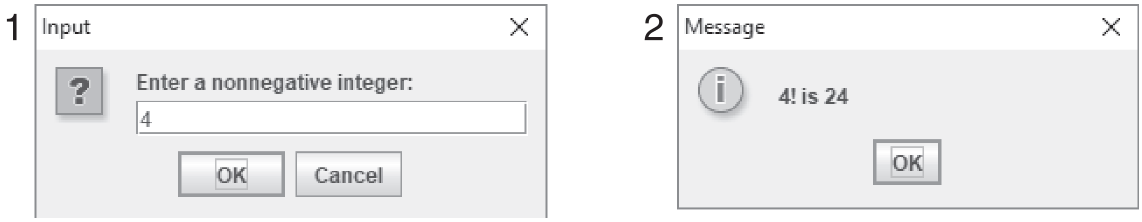
```
private static int factorial(int n)
{
   if (n == 0)
      return 1;   // Base case
   else
      return n * factorial(n - 1);
}
```

The program in Code Listing 15-4 demonstrates the method. Figure 15-4 shows example interaction with the program.

**Code Listing 15-4** (FactorialDemo.java)

```
import javax.swing.JOptionPane;

/**
   This program demonstrates the recursive
   factorial method.
*/

public class FactorialDemo
{
   public static void main(String[] args)
   {
      String input;  // To hold user input
      int number;    // To hold a number

      // Get a number from the user.
      input = JOptionPane.showInputDialog("Enter a " +
                                    "nonnegative integer:");
      number = Integer.parseInt(input);

      // Display the factorial of the number.
      JOptionPane.showMessageDialog(null,
                  number + "! is " + factorial(number));

      System.exit(0);
   }

   /**
      The factorial method uses recursion to calculate
      the factorial of its argument, which is assumed
      to be a nonnegative number.
      @param n The number to use in the calculation.
      @return The factorial of n.
   */

   private static int factorial(int n)
   {
      if (n == 0)
         return 1;    // Base case
      else
         return n * factorial(n - 1);
   }
}
```

**Figure 15-4** Interaction with the `FactorialDemo.java` program (Oracle Corporate Counsel)



Enter a nonnegative integer: 4

4! is 24

In the example run of the program, the factorial method is called with the argument 4 passed into `n`. Because `n` is not equal to 0, the `if` statement's `else` clause executes the following statement:

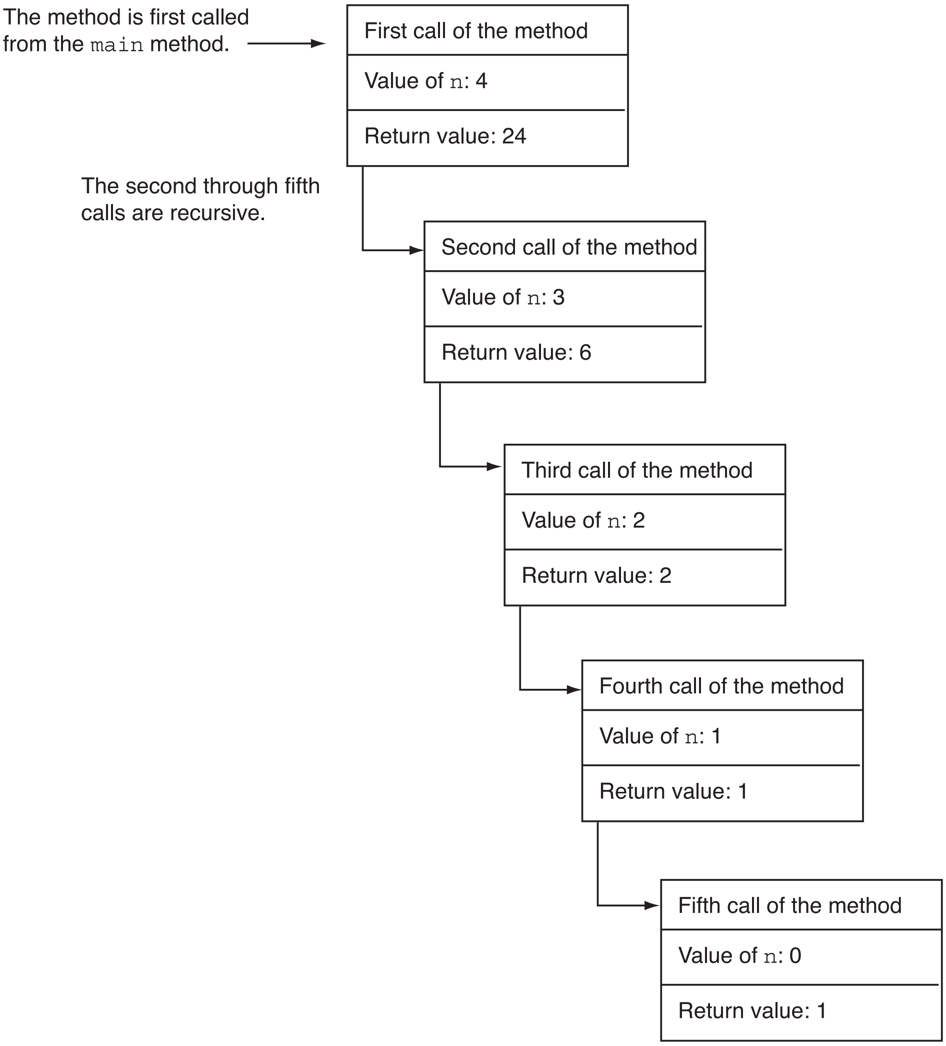
```
return n * factorial(n - 1);
```

Although this is a `return` statement, it does not immediately return. Before the return value can be determined, the value of `factorial(n - 1)` must be determined. The `factorial` method is called recursively until the fifth call, in which the `n` parameter will be set to zero. The diagram in Figure 15-5 illustrates the value of `n` and the return value during each call of the method.

This diagram illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually the recursion has to stop in order for a solution to be reached. If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our `factorial` method, the value of the parameter `n` gets closer to 0 with each recursive call. When the parameter reaches 0, the method returns a value without making another recursive call.

**Figure 15-5** Recursive calls to the factorial method (Oracle Corporate Counsel)


The method is first called from the main method.
The second through fifth calls are recursive.
First call of the method
Value of n: 4
Return value: 24
Second call of the method
Value of n: 3
Return value: 6
Third call of the method
Value of n: 2
Return value: 2
Fourth call of the method
Value of n: 1
Return value: 1
Fifth call of the method
Value of n: 0
Return value: 1

## Direct and Indirect Recursion

The examples we have discussed so far show recursive methods that directly call themselves. This is known as direct recursion. There is also the possibility of creating indirect recursion in a program. This occurs when method A calls method B, which in turn calls method A. There can even be several methods involved in the recursion. For example, method A could call method B, which could call method C, which calls method A.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

15.1 It is said that a recursive algorithm has more overhead than an iterative algorithm. What does this mean?

15.2 What is a base case?

15.3 What is a recursive case?

15.4 What causes a recursive algorithm to stop calling itself?

15.5 What is direct recursion? What is indirect recursion?

# 15.3 Examples of Recursive Methods

## Summing a Range of Array Elements with Recursion

In this example we look at a method, `rangeSum`, that uses recursion to sum a range of array elements. The method takes the following arguments: an `int` array that contains the range of elements to be summed, an `int` specifying the starting element of the range, and an `int` specifying the ending element of the range. Here is an example of how the method might be used:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int sum;
sum = rangeSum(numbers, 3, 7);
```

This code specifies that `rangeSum` should return the sum of elements 3 through 7 in the `numbers` array. The return value, which in this case would be 30, is stored in `sum`. Here is the definition of the `rangeSum` method:

```
public static int rangeSum(int[] array, int start, int end)
{
   if (start > end)
      return 0;
   else
      return array[start] + rangeSum(array, start + 1, end);
}
```

This method's base case is when the `start` parameter is greater than the `end` parameter. If this is true, the method returns the value `0`. Otherwise, the method executes the following statement:

```
return array[start] + rangeSum(array, start + 1, end);
```

This statement returns the sum of `array[start]` plus the return value of a recursive call. Notice that in the recursive call, the starting element in the range is `start + 1`. In essence, this statement says "return the value of the first element in the range plus the sum of the rest of the elements in the range." The program in Code Listing 15-5 demonstrates the method.

**Code Listing 15-5** (`RangeSum.java`)

```
/**
   This program demonstrates the recursive rangeSum method.
*/

```

```
public class RangeSum
{

   public static void main(String[] args)
   {
      int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

      System.out.print("The sum of elements 2 through " +
                       "5 is "+ rangeSum(numbers, 2, 5));
   }

   /**
      The rangeSum method calculates the sum of a specified
      range of elements in array.
      @param start Specifies the starting element.
      @param end Specifies the ending element.
      @return The sum of the range.
   */

   public static int rangeSum(int[] array, int start, int end)
   {
      if (start > end)
         return 0;
      else
         return array[start] +
                rangeSum(array, start + 1, end);
   }
}
```

**Program Output**

```
The sum of elements 2 through 5 is 18
```

## Drawing Concentric Circles

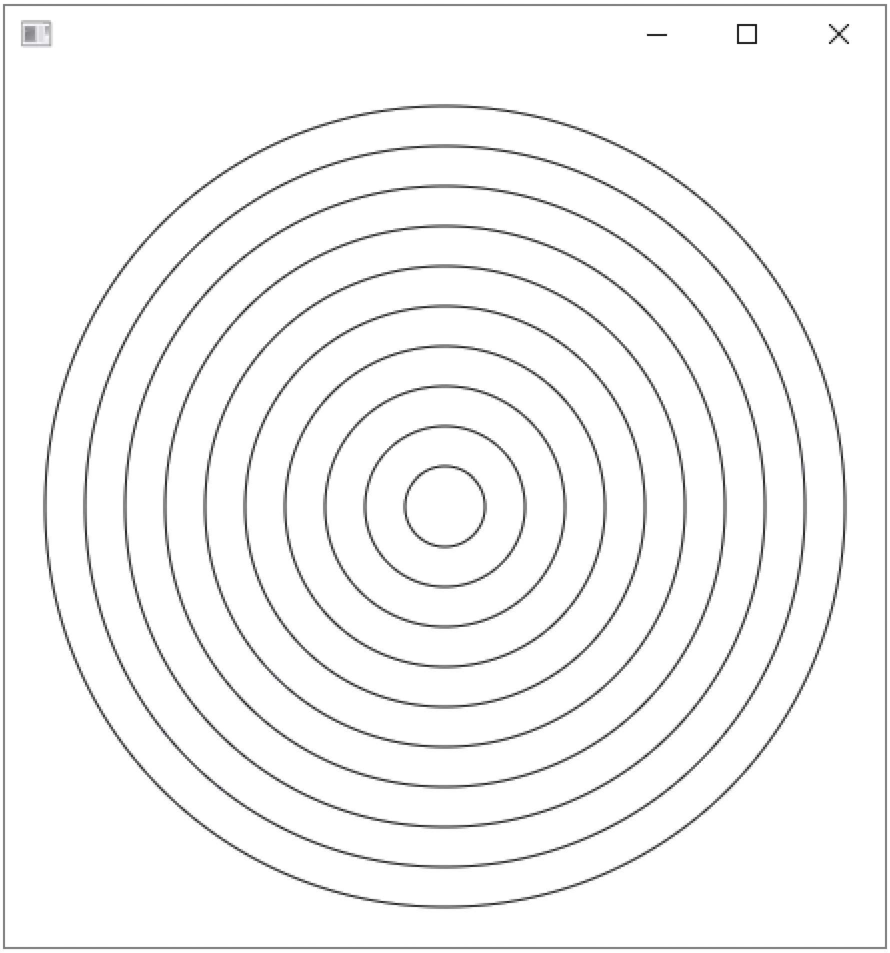
In this example we look at the `Circles` application, which uses recursion to draw concentric circles. Concentric circles are circles that share a common center point. Figure 15-6 shows the application's output. The code is shown in Code Listing 15-6.

**Code Listing 15-6** (`Circles.java`)

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
```

```

public class Circles extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Constants for the scene size
      final double SCENE_WIDTH = 440.0;
      final double SCENE_HEIGHT = 440.0;

      // Constants for the starting values
      final double CENTER_X = 220.0;
      final double CENTER_Y = 220.0;
      final double RAD = 20.0;
      final int NUM_CIRCLES = 10;

      // Create an empty Pane.
      Pane pane = new Pane();

      // Recursively add 10 circles to the Pane.
      drawCircles(pane, NUM_CIRCLES, CENTER_X, CENTER_Y, RAD);

      // Create a Scene and display it.
      Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
      primaryStage.setScene(scene);
      primaryStage.show();
   }

   /**
      The drawCircles method draws concentric circles.
      It accepts the following arguments:
      p -- a Pane object to add the circles to
      n -- the number of circles to draw
      x -- x coordinate of the circle's center point
      y -- y coordinate of the circle's center point
      rad -- the circle's radius
   */

   private void drawCircles(Pane p, int n, double x, double y, double rad)
   {
      if (n > 0)
      {
         Circle circle = new Circle(x, y, rad);  // Create the circle
55            circle.setStroke(Color.BLACK);          // Line color is black
56            circle.setFill(null);                   // No fill color
57            p.getChildren().addAll(circle);         // Add the it to the Pane
58            drawCircles(p, n - 1, x, y, rad + 20);  // Draw the next circle
59         }
60     }
61  }
```

**Figure 15-6** Circles applet (Oracle Corporate Counsel)



The `drawCircles` method, which is called from the application's `start` method, uses recursion to draw the concentric circles. The `n` parameter holds the number of circles to draw. If this parameter is set to 0, the method has reached its base case. Otherwise, line 54 creates a `Circle` object at the specified center point, using the specified radius. Line 55 sets the `Circle`'s stroke color to black, and line 56 sets the `Circle`'s fill color to `null` (the circle will not be filled with a color). Line 57 adds the `Circle` object to the `Pane`. Line 58 makes a recursive call to the `drawCircles` method with the parameters adjusted for the next circle.

## The Fibonacci Series

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of Fibonacci numbers. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, . . .

Notice that after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as follows:

If $n = 0$ then $\text{Fib}(n) = 0$
If $n = 1$ then $\text{Fib}(n) = 1$
If $n >= 2$ then $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$

A recursive Java method to calculate the *n*th number in the Fibonacci series is shown here:

```
public static int fib(int n)
{
   if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fib(n - 1) + fib(n - 2);
}
```

Notice that this method actually has two base cases: when `n` is equal to 0 and when `n` is equal to 1. In either case, the method returns a value without making a recursive call. The program in Code Listing 15-7 demonstrates this method by displaying the first 10 numbers in the Fibonacci series.

**Code Listing 15-7** (FibNumbers.java)

```
/**
   This program demonstrates the recursive fib method.
*/

public class FibNumbers
{
   public static void main(String[] args)
   {
      System.out.println("The first 10 numbers in " +
                         "the Fibonacci series are:");

      for (int i = 0; i < 10; i++)
         System.out.print(fib(i) + " ");

      System.out.println();
   }

   /**
      The fib method calculates the nth
      number in the Fibonacci series.
      @param n The nth number to calculate.
      @return The nth number.
   */

   public static int fib(int n)
   {
      if (n == 0)
         return 0;
      else if (n == 1)
         return 1;
      else
         return fib(n - 1) + fib(n - 2);
   }
}
```

**Program Output**

```
The first 10 numbers in the Fibonacci series are:
0 1 1 2 3 5 8 13 21 34
```

## Finding the Greatest Common Divisor

Our next example of recursion is the calculation of the greatest common divisor, or GCD, of two numbers. The GCD of two positive integers, $x$ and $y$, is as follows:

if $y$ divides $x$ evenly, then $\gcd(x, y) = y$
Otherwise, $\gcd(x, y) = \gcd(y, \text{remainder of } x/y)$

This definition states that the GCD of $x$ and $y$ is $y$ if $x/y$ has no remainder. This is the base case. Otherwise, the answer is the GCD of $y$ and the remainder of $x/y$. The program in Code Listing 15-8 shows a recursive method for calculating the GCD.

**Code Listing 15-8** (`GCDdemo.java`)

```
import java.util.Scanner;

/**
   This program demonstrates the recursive gcd method.
*/

public class GCDdemo
```

```
{
   public static void main(String[] args)
   {
      int num1, num2; // Two numbers for GCD calculation

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the first number from the user.
      System.out.print("Enter an integer: ");
      num1 = keyboard.nextInt();

      // Get the second number from the user.
      System.out.print("Enter another integer: ");
      num2 = keyboard.nextInt();

      // Display the GCD.
      System.out.println("The greatest common divisor " +
                         "of these two numbers is " +
                         gcd(num1, num2));
   }

   /**
      The gcd method calculates the greatest common
      divisor of the arguments passed into x and y.
      @param x A number.
      @param y Another number.
      @returns The greatest common divisor of x and y.
   */

   public static int gcd(int x, int y)
   {
      if (x % y == 0)
         return y;
      else
         return gcd(y, x % y);
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter an integer: 49 [Enter]
Enter another integer: 28 [Enter]
The greatest common divisor of these two numbers is 7
```

## 15.4 A Recursive Binary Search Method

**CONCEPT: The recursive binary search algorithm is more elegant and easier to understand than its iterative version.**

In Chapter 7 you learned about the binary search algorithm and saw an iterative example written in Java. The binary search algorithm can also be implemented recursively. For example, the procedure can be expressed as:

```
If array[middle] equals the search value, then the value is found.
Else if array[middle] is less than the search value, perform
    a binary search on the upper half of the array.
Else if array[middle] is greater than the search value, perform
    a binary search on the lower half of the array.
```

When you compare the recursive algorithm to its iterative counterpart, it becomes evident that the recursive version is much more elegant and easier to understand. The recursive binary search algorithm is also a good example of repeatedly breaking a problem down into smaller pieces until it is solved. Here is the code for the method:

```
public static int binarySearch(int[] array, int first,
                               int last, int value)
{
   int middle;     // Mid point of search
   // Test for the base case where the
   // value is not found.
   if (first > last)
      return -1;
   // Calculate the middle position.
   middle = (first + last) / 2;
   // Search for the value.
   if (array[middle] == value)
      return middle;
   else if (array[middle] < value)
      return binarySearch(array, middle + 1,
                          last, value);
   else
      return binarySearch(array, first,
                          middle - 1, value);
}
```

The first parameter, `array`, is the array to be searched. The next parameter, `first`, holds the subscript of the first element in the search range (the portion of the array to be searched). The next parameter, `last`, holds the subscript of the last element in the search range. The last parameter, `value`, holds the value to be searched for. Like the iterative version, this method returns the subscript of the value if it is found, or −1 if the value is not found. Code Listing 15-9 demonstrates the method.

**Code Listing 15-9** (RecursiveBinarySearch.java)

```
import java.util.Scanner;

/**
   This program demonstrates the recursive
   binary search method.
*/

public class RecursiveBinarySearch
{
   public static void main(String [] args)
   {
      int searchValue;   // The value to search for
      int result;        // The search result
      String input;      // A line of input
      char again;        // To hold a single character

      // The values in the following array are sorted
      // in ascending order.
      int numbers[] = {101, 142, 147, 189, 199, 207, 222,
                       234, 289, 296, 310, 319, 388, 394,
                       417, 429, 447, 521, 536, 600};

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      do
      {
         // Get a value to search for.
         System.out.print("Enter a value to " +
                          "search for: ");
         searchValue = keyboard.nextInt();

         // Search for the value
         result = binarySearch(numbers, 0,
                                 (numbers.length - 1),
                                 searchValue);

         // Display the results.
         if (result == -1)
         {
            System.out.println(searchValue +
                               " was not found.");
         }
         else
         {
            System.out.println(searchValue +
                               " was found at " +
                               "element " + result);
         }

         // Does the user want to search again?
         System.out.print("Do you want to search again? " +
                          "(Y or N): ");
         // Consume the remaining newline.
         keyboard.nextLine();
         // Read a line of input.
         input = keyboard.nextLine();

      } while (input.charAt(0) == 'y' ||
               input.charAt(0) == 'Y');
   }

   /**
      The binarySearch method performs a binary search
      on an integer array.
      @param array The array to search.
      @param first The first element in the search range.
      @param last The last element in the search range.
      @param value The value to search for.
      @return The subscript of the value if found,
              otherwise -1.
   */

   public static int binarySearch(int[] array, int first,
                                  int last, int value)
   {
      int middle;     // Mid point of search

      // Test for the base case where the
      // value is not found.
      if (first > last)
         return -1;

      // Calculate the middle position.
      middle = (first + last) / 2;

      // Search for the value.
      if (array[middle] == value)
         return middle;
      else if (array[middle] < value)
```

```
            return binarySearch(array, middle + 1,
                                    last, value);
         else
            return binarySearch(array, first,
                                    middle - 1, value);
      }
   }
```

**Program Output with Example Input Shown in Bold**

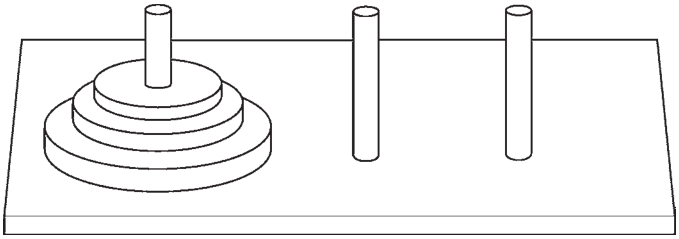
```
Enter a value to search for: 289 [Enter]
289 was found at element 8
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 388 [Enter]
388 was found at element 12
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 101 [Enter]
101 was found at element 0
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 999 [Enter]
999 was not found.
Do you want to search again? (Y or N): n [Enter]
```

## 15.5 The Towers of Hanoi

**CONCEPT: The repetitive steps involved in solving the Towers of Hanoi game can be easily implemented in a recursive algorithm.**

The Towers of Hanoi is a mathematical game that is often used in computer science textbooks to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in Figure 15-7.

**Figure 15-7** The pegs and discs in the Towers of Hanoi game



Notice that the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend where a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The job of the monks is to move the discs from the first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disk may be moved at a time.
- A disk cannot be placed on top of a smaller disc.
- All discs must be stored on a peg except while being moved.

According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you have only one disc, the solution to the game is simple: move the disc from peg 1 to peg 3. If you have two discs, the solution requires three moves:

- Move disc 1 to peg 2.
- Move disc 2 to peg 3.
- Move disc 1 to peg 3.

Notice that this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increases. To move three discs requires the seven moves shown in Figure 15-8.

The following statement describes the overall solution to the problem:

*Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.*

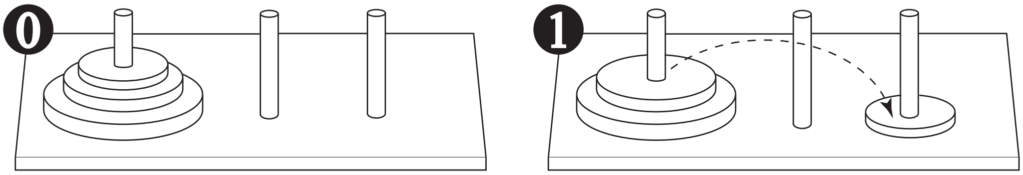
The following algorithm can be used as the basis of a recursive method that simulates the solution to the game. Notice that in this algorithm we use the variables A, B, and C to hold peg numbers.

```
To move n discs from peg A to peg C, using peg B as a temporary peg:
If n > 0 then
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.
    Move the remaining disc from peg A to peg C.
    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.
End if
```

The base case for the algorithm is reached when there are no more discs to move. The following code is for a method that implements this algorithm. Note that the method does not actually move anything, but displays instructions indicating all of the disc moves to make.

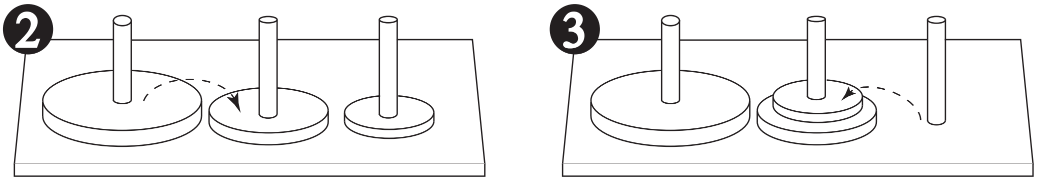
```
private void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
   if (num > 0)
   {
      moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
      System.out.println("Move a disc from peg " + fromPeg +
                         " to peg " + toPeg);
      moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
   }
}
```

**Figure 15-8** Steps for moving three pegs



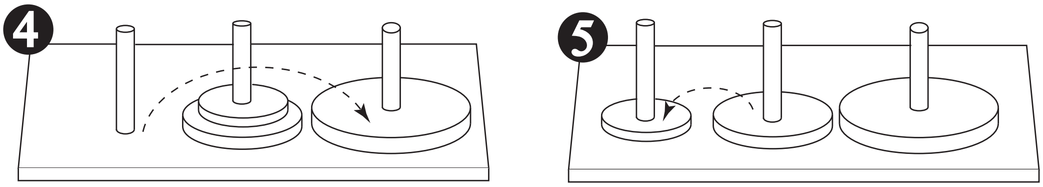
Original setup.

First move: Move disc 1 to peg 3.



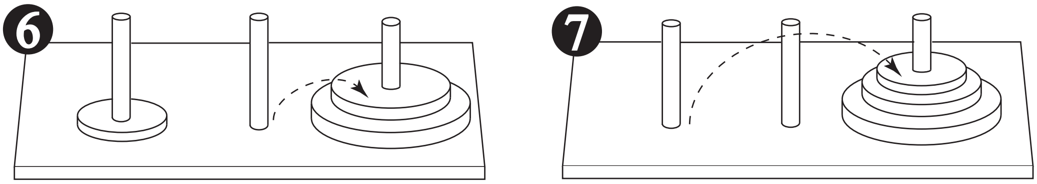
Second move: Move disc 2 to peg 2.

Third move: Move disc 1 to peg 2.



Fourth move: Move disc 3 to peg 3.

Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.

Seventh move: Move disc 1 to peg 3.

This method accepts arguments into the following four parameters:

| | |
|---|---|
| `num` | The number of discs to move. |
| `fromPeg` | The peg to move the discs from. |
| `toPeg` | The peg to move the discs to. |
| `tempPeg` | The peg to use as a temporary peg. |

If `num` is greater than 0, then there are discs to move. The first recursive call is as follows:

```
moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
```

This statement is an instruction to move all but one disc from `fromPeg` to `tempPeg`, using `toPeg` as a temporary peg. The next statement is as follows:

```
System.out.println("Move a disc from peg " + fromPeg +
                   " to peg " + toPeg);
```

This simply displays a message indicating that a disc should be moved from `fromPeg` to `toPeg`. Next, another recursive call is executed as follows:

```
moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
```

This statement is an instruction to move all but one disc from `tempPeg` to `toPeg`, using `fromPeg` as a temporary peg. Code Listing 15-10 shows the `Hanoi` class, which uses this method.

**Code Listing 15-10** (`Hanoi.java`)

```
/**
   This class displays a solution to the Towers of
   Hanoi game.
*/

public class Hanoi
{
   private int numDiscs;   // Number of discs

   /**
      Constructor.
      @param n The number of discs to use.
   */

   public Hanoi(int n)
   {
      // Assign the number of discs.
      numDiscs = n;

      // Move the number of discs from peg 1 to peg 3
      // using peg 2 as a temporary storage location.
      moveDiscs(numDiscs, 1, 3, 2);
   }

   /**
      The moveDiscs method displays a disc move.
      @param num The number of discs to move.
      @param fromPeg The peg to move from.
      @param toPeg The peg to move to.
      @param tempPeg The temporary peg.
   */

   private void moveDiscs(int num, int fromPeg,
                          int toPeg, int tempPeg)
   {
      if (num > 0)
37       {
38          moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
39          System.out.println("Move a disc from peg " +
40                             fromPeg + " to peg " + toPeg);
41          moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
42     }
43   }
44 }
```

The class constructor accepts an argument, which is the number of discs to use in the game. It assigns this value to the `numDiscs` field, and then calls the `moveDiscs` method in line 22. In a nutshell, this statement is an instruction to move all the discs from peg 1 to peg 3, using peg 2 as a temporary peg. The program in Code Listing 15-11 demonstrates the class. It displays the instructions for moving three discs.

**Code Listing 15-11** (`HanoiDemo.java`)

```
/**
   This class demonstrates the Hanoi class, which
   displays the steps necessary to solve the Towers
   of Hanoi game.
*/

public class HanoiDemo
{
   static public void main(String[] args)
   {
      Hanoi towersOfHanoi = new Hanoi(3);
   }
}
```

**Program Output**

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
```

## 15.6 Common Errors to Avoid

- **Not coding a base case.** When the base case is reached, a recursive method stops calling itself. Without a base case, the method will continue to call itself infinitely.
- **Not reducing the problem with each recursive call.** Unless the problem is reduced (which usually means that the value of one or more critical parameters is reduced)

with each recursive call, the method will not reach the base case. If the base case is not reached, the method will call itself infinitely.

- **Writing the recursive call in such a way that the base case is never reached.** You might have a base case and a recursive case that reduces the problem, but if the calculations are not performed in such a way that the base case is ultimately reached, the method will call itself infinitely.

## Review Questions and Exercises

### Multiple Choice and True/False

1. A method is called once from a program's `main` method, and then it calls itself four times. The depth of recursion is __________.
   a. one
   b. four
   c. five
   d. nine
2. This is the part of a problem that can be solved without recursion.
   a. base case
   b. solvable case
   c. known case
   d. iterative case
3. This is the part of a problem that is solved with recursion.
   a. base case
   b. iterative case
   c. unknown case
   d. recursion case
4. This is when a method explicitly calls itself.
   a. explicit recursion
   b. modal recursion
   c. direct recursion
   d. indirect recursion
5. This is when method A calls method B, which calls method A.
   a. implicit recursion
   b. modal recursion
   c. direct recursion
   d. indirect recursion
6. This refers to the actions taken internally by the JVM when a method is called.
   a. overhead
   b. set up
   c. clean up
   d. synchronization
7. **True or False:** An iterative algorithm will usually run faster than an equivalent recursive algorithm.
8. **True or False:** Some problems can be solved through recursion only.

9. **True or False:** It is not necessary to have a base case in all recursive algorithms.
10. **True or False:** In the base case, a recursive method calls itself with a smaller version of the original problem.

### Find the Error

1. Find the error in the following program:

```
public class FindTheError
{
   public static void main(String[] args)
   {
      myMethod(0);
   }
   public static void myMethod(int num)
   {
      System.out.print(num + " ");
      myMethod(num + 1);
   }
}
```

### Algorithm Workbench

1. Write a method that accepts a `String` as an argument. The method should use recursion to display each individual character in the `String`.
2. Modify the method you wrote in Algorithm Workbench 1 so it displays the `String` backwards.
3. What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      int num = 0;
      showMe(num);
   }
   public static void showMe(int arg)
   {
      if (arg < 10)
         showMe(arg + 1);
      else
         System.out.println(arg);
   }
}
```

4. What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      int num = 0;
      showMe(num);
   }
   public static void showMe(int arg)
   {
      System.out.println(arg);
      if (arg < 10)
         showMe(arg + 1);
   }
}
```

5. What will the following program display?

```
public class ReviewQuestion5
{
   public static void main(String[] args)
   {
      int x = 10;
      System.out.println(myMethod(x));
   }
   public static int myMethod(int num)
   {
      if (num <= 0)
         return 0;
      else
         return myMethod(num − 1) + num;
  }
}
```

6. Convert the following iterative method to one that uses recursion:

```
public static void sign(int n)
{
   while (n > 0)
   {
      System.out.println("No Parking");
      n--;
   }
}
```

7. Write an iterative version (using a loop instead of recursion) of the `factorial` method shown in this chapter.

### Short Answer

1. What is the difference between an iterative algorithm and a recursive algorithm?
2. What is a recursive algorithm's base case? What is the recursive case?
3. What is the base case of each of the recursive methods listed in Algorithm Workbench 3, 4, and 5?
4. What type of recursive method do you think would be more difficult to debug: one that uses direct recursion or one that uses indirect recursion? Why?
5. Which repetition approach is less efficient: a loop or a recursive method? Why?
6. When recursion is used to solve a problem, why must the recursive method call itself to solve a smaller version of the original problem?
7. How is a problem usually reduced with a recursive method?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Recursive Multiplication

Write a recursive function that accepts two arguments into the parameters `x` and `y`. The function should return the value of `x` times `y`. Remember, multiplication can be performed as repeated addition as follows:

$$7 * 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4$$

### 2. `isMember` Method

Write a recursive `boolean` method named `isMember`. The method should search an array for a specified value, and return `true` if the value is found in the array, or `false` if the value is not found in the array. Demonstrate the method in a program.

### 3. String Reverser

Write a recursive method that accepts a string as its argument and prints the string in reverse order. Demonstrate the method in a program.

### 4. `maxElement` Method

Write a method named `maxElement`, which returns the largest value in an array that is passed as an argument. The method should use recursion to find the largest element. Demonstrate the method in a program.

### 5. Palindrome Detector

A palindrome is any word, phrase, or sentence that reads the same forward and backward. Here are some well-known palindromes:

Able was I, ere I saw Elba
A man, a plan, a canal, Panama
Desserts, I stressed
Kayak

Write a `boolean` method that uses recursion to determine whether a `String` argument is a palindrome. The method should return `true` if the argument reads the same forward and backward. Demonstrate the method in a program.

### 6. Character Counter

Write a method that uses recursion to count the number of times a specific character occurs in an array of characters. Demonstrate the method in a program.

VideoNote
The Recursive Power Problem

### 7. Recursive Power Method

Write a method that uses recursion to raise a number to a power. The method should accept two arguments: the number to be raised and the exponent. Assume that the exponent is a nonnegative integer. Demonstrate the method in a program.

### 8. Sum of Numbers

Write a method that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the method will return the sum of 1, 2, 3, 4, . . . 50. Use recursion to calculate the sum. Demonstrate the method in a program.

### 9. Ackermann's Function

Ackermann's function is a recursive mathematical algorithm that can be used to test how well a computer performs recursion. Write a method `ackermann(m, n)`, which solves Ackermann's function. Use the following logic in your method:

If `m` = 0 then return n + 1
If `n` = 0 then return `ackermann(m - 1, 1)`
Otherwise, return `ackermann(m - 1, ackermann(m, n - 1))`

Test your method in a program that displays the return values of the following method calls:

```
ackermann(0, 0)  ackermann(0, 1)  ackermann(1, 1)  ackermann(1, 2)
ackermann(1, 3)  ackermann(2, 2)  ackermann(3, 2)
```

### 10. Recursive Population Class

In Programming Challenge 9 of Chapter 4 you wrote a population class that predicts the size of a population of organisms after a number of days. Modify the class so it uses a recursive method instead of a loop to calculate the number of organisms.

CHAPTER

# 16 Sorting, Searching, and Algorithm Analysis

TOPICS

16.1 Introduction to Sorting Algorithms
16.2 Introduction to Search Algorithms
16.3 Analysis of Algorithms
16.4 Common Errors to Avoid

## 16.1 Introduction to Sorting Algorithms

**CONCEPT: Sorting algorithms are used to arrange data into some order.**

Often, the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Product codes could be sorted so all the products of the same color are stored together. To sort the data in an array, the programmer must use an appropriate *sorting algorithm*. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order.

The data in an array can be sorted in either ascending or descending order. If an array is sorted in *ascending order*, it means the values in the array are stored from lowest to highest. If the values are sorted in *descending order*, they are stored from highest to lowest. This section will introduce four sorting algorithms that can be used to sort the data in an array: the *bubble sort*, the *selection sort*, the *insertion sort*, and the *Quicksort*.

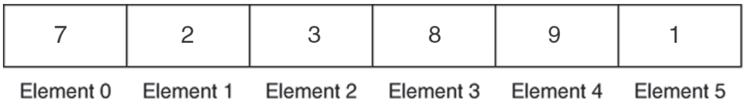
### The Bubble Sort

VideoNote
The Bubble Sort Algorithm

The bubble sort is an easy way to arrange data in ascending or descending order. In this section we will see how the bubble sort can be used to sort an array in ascending order. This particular algorithm is called the *bubble sort* because it makes several passes through the elements of the array and the larger values "bubble" toward the end of the array with each pass.

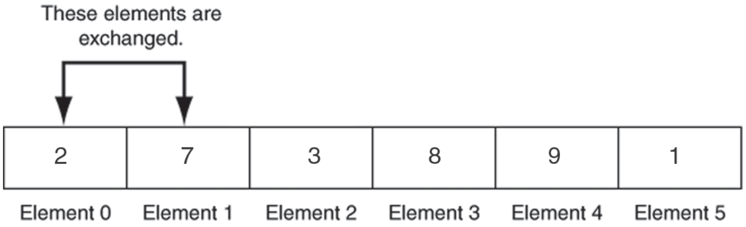
Suppose we have the array shown in Figure 16-1. Let's see how the bubble sort can be used in arranging the array's elements in ascending order.

**Figure 16-1** An array


7
2
3
8
9
1
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

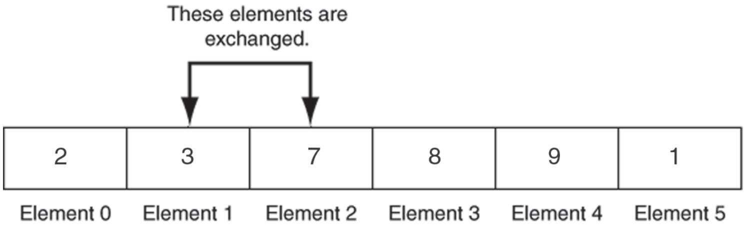
The bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are exchanged. After the exchange, the array would appear as shown in Figure 16-2.

**Figure 16-2** Elements 0 and 1 are exchanged


These elements are exchanged.
2
7
3
8
9
1
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

This step is repeated with elements 1 and 2. If element 1 is greater than element 2, they are exchanged. The array would then appear as shown in Figure 16-3.

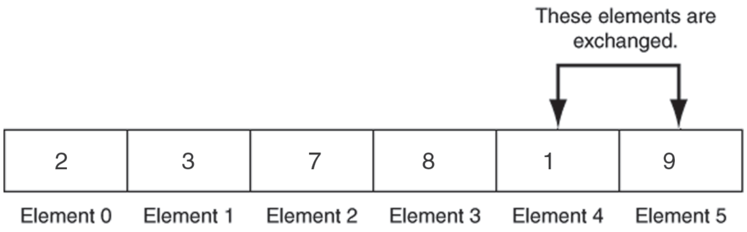
**Figure 16-3** Elements 1 and 2 are exchanged


These elements are exchanged.
2
3
7
8
9
1
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

Next, elements 2 and 3 are compared. In this array, these elements are already in the proper order (element 2 is less than element 3), so no exchange takes place. As the cycle continues, elements 3 and 4 are compared. Once again, no exchange is necessary because they are already in the proper order.

When elements 4 and 5 are compared, however, an exchange must take place because element 4 is greater than element 5. The array now appears as shown in Figure 16-4.

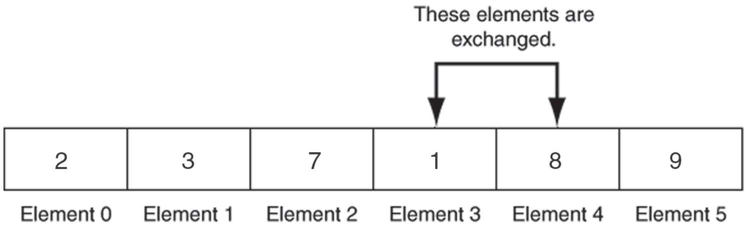
**Figure 16-4** Elements 4 and 5 are exchanged


These elements are exchanged.
2
3
7
8
1
9
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

At this point, the entire array has been scanned, and the largest value, 9, is in the correct position. There are other elements, however, that are not yet in their final positions. So, we make another pass through the array, comparing each element with its neighbor. In the next pass we can stop comparing after we reach the next-to-last element because the last element is already at the correct position.

The second pass starts by comparing elements 0 and 1. Because those two are in the proper order, no exchange takes place. Elements 1 and 2 are compared next, but once again, no exchange takes place. This continues until elements 3 and 4 are compared. Because element 3 is greater than element 4, they are exchanged. Element 4 is the last element that we will compare during this pass, so this pass stops. The array now appears as shown in Figure 16-5.

**Figure 16-5** Elements 3 and 4 are exchanged


These elements are exchanged.
2
3
7
1
8
9
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

At the end of the second pass, the last two elements in the array are in their correct positions. The third pass starts now, comparing each element with its neighbor. The third pass will not involve the last two elements, however, because they are already in their correct positions. When the third pass is finished, the last three elements will also have been correctly placed, as shown in Figure 16-6.

Each time the algorithm makes a pass through the array, the portion of the array that is scanned is decreased in size by one element, and the largest value in the scanned portion of the array is moved to its final position. When all of the passes have been made, the array will appear as shown in Figure 16-7.

**Figure 16-6** The array after the third pass

| | | | | | |
|---|---|---|---|---|---|
| 2 | 3 | 1 | 7 | 8 | 9 |
| Element 0 | Element 1 | Element 2 | Element 3 | Element 4 | Element 5 |

**Figure 16-7** The array with all elements sorted

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 7 | 8 | 9 |
| Element 0 | Element 1 | Element 2 | Element 3 | Element 4 | Element 5 |

The following pseudocode shows the logic of the bubble sort.

```
For lastPosition = last subscript subscript in the array, decremented down to 0
    For index = 0 up through lastPosition − 1
        If array[index] > array[index + 1]
            Swap array[index] with array[index + 1]
        End If
    End For
End For
```

Code Listing 16-1 shows the `IntBubbleSorter` class, which has a public static method, `bubbleSort`. You pass an `int` array to the method and it uses the bubble sort algorithm to sort the array's contents in ascending order.

**Code Listing 16-1** `(IntBubbleSorter.java)`

```
/**
   The IntBubbleSorter class provides a public static
   method for performing a bubble sort on an int array.
*/

public class IntBubbleSorter
{

   /**
      The bubbleSort method uses the bubble sort algorithm
      to sort an int array.
      @param array The array to sort.
   */

   public static void bubbleSort(int[] array)
   {
      int lastPos;   // Position of the last element to compare
      int index;     // Index of an element to compare
19       int temp;     // Used to swap to elements
20
21       // The outer loop positions lastPos at the last element
22       // to compare during each pass through the array. Initially
23       // lastPos is the index of the last element in the array.
24       // During each iteration, it is decreased by one.
25       for (lastPos = array.length – 1; lastPos >= 0; lastPos--)
26       {
27          // The inner loop steps through the array, comparing
28          // each element with its neighbor. All of the elements
29          // from index 0 through lastPos are involved in the
30          // comparison. If two elements are out of order, they
31          // are swapped.
32          for (index = 0; index <= lastPos – 1; index++)
33          {
34             // Compare an element with its neighbor.
35             if (array[index] > array[index + 1])
36             {
37                // Swap the two elements.
38                temp = array[index];
39                array[index] = array[index + 1];
40                array[index + 1] = temp;
41             }
42          }
43       }
44    }
45 }
```

Lines 17, 18, and 19 declare three variables: `lastPos`, `index`, and `temp`. Here are the purposes of the variables:

- The `lastPos` variable will hold the index of the last element that is to be compared to its neighbor during a pass through the array.
- The `index` variable is used as an index into the array during each pass.
- The `temp` variable is used to hold the value of an element temporarily during a swap.

The method uses two `for` loops, one nested inside another. The outer loop begins in line 25 as follows:

```
for (lastPos = array.length – 1; lastPos >= 0; lastPos--)
```

This loop will iterate once for each element in the array. It causes the `lastPos` variable to take on all of the array's subscripts, from the highest subscript down to 0. After each iteration, `lastPos` is decremented by one.

The second loop, which is nested inside the first loop, begins in line 32 as follows:

```
for (index = 0; index <= lastPos – 1; index++)
```

This loop iterates once for each element in the portion of the array that is still unsorted. It starts `index` at 0 and increments it up through `lastPos - 1`. During each iteration, the comparison in line 35 is performed:

```
if (array[index] > array[index + 1])
```

This `if` statement compares the element at `array[index]` with its neighbor `array[index + 1]`. If the element's neighbor is less, then the two are swapped by the code in lines 38 through 40.

The program in Code Listing 16-2 tests the method.

**Code Listing 16-2** `(BubbleSortTest.java)`

```java
/**
   This program tests the bubbleSort method in the
   IntBubbleSorter class.
*/

public class BubbleSortTest
{
   public static void main(String[] args)
   {
      // Create an int array with test values.
      int[] values = { 5, 1, 3, 6, 4, 2 };

      // Display the array's contents.
      System.out.println("Original order: ");
      for (int element : values)
         System.out.print(element + " ");

      // Sort the array.
      IntBubbleSorter.bubbleSort(values);

      // Display the array's contents.
      System.out.println("\nSorted order: ");
      for (int element : values)
         System.out.print(element + " ");

      System.out.println();
   }
}
```

**Program Output**

```
Original order:
5 1 3 6 4 2
Sorted order:
1 2 3 4 5 6
```

## Using the Bubble Sort to Sort Objects

The previous example demonstrated how the bubble sort algorithm can be used to sort an array of `int` values. But, suppose we have an array of objects that we wish to sort. How could we modify the `bubbleSort` method shown previously so it can work with objects?

To sort an array of objects, we must have a consistent way to compare the objects. In the `IntBubbleSorter` class, line 35 of the `bubbleSort` method compares two elements of the array and determines whether one is greater than the other. Because that method is comparing primitive `int` values, it simply compares the two elements with the greater than (>) operator. This approach will not work with object references, however. Reference variables hold only the addresses of objects, and you cannot use operators like greater than (>) or less than (<) to compare them. (Even if you could use these operators with object references, it would not make sense to do so. When you compare two objects, you usually want to compare the data contained within those objects.)

The Java API has an interface named `Comparable`, which helps in a situation like this. The `Comparable` interface, which is in the `java.lang` package, specifies a method named `compareTo`, which is used to compare the calling object with another object that is passed as an argument. The method returns an `int` value that specifies whether the calling object is greater than (>), less than (<), or equal to (=) the object that was passed as an argument.

Many of the classes in the Java API implement this interface, including `String` and the wrapper classes. You might recall from our discussion of the `String` class in Chapter 3 how the `compareTo` method works:

- If the calling object is less than the object passed as an argument, the method returns a negative number.
- If the calling object is equal to the object passed as an argument, the method returns 0.
- If the calling object is greater than the object passed as an argument, the method returns a positive number.

If we want to write a `bubbleSort` method that can accept an array of objects, a good solution is to require that the objects in the array implement the `Comparable` interface. That way we can use the `compareTo` method to compare the elements. Code Listing 16-3 shows the `ObjectBubbleSorter` class, which has a `bubbleSort` method that sorts an array of `Comparable` objects.

**Code Listing 16-3** (`ObjectBubbleSorter.java`)

```
/**
   The ObjectBubbleSorter class provides a public static
   method for performing a bubble sort on an array of
   objects implementing the Comparable interface.
*/

public class ObjectBubbleSorter
{

   /**
      The bubbleSort method uses the bubble sort algorithm
      to sort an array of objects implementing Comparable
      @param array The array to sort.
   */

   public static void bubbleSort(Comparable[] array)
   {
      int lastPos;     // Marks the last position to compare
      int index;       // Index of an element to compare
      Comparable temp; // Used to swap to elements

      // The outer loop positions lastPos at the last element
      // to compare during each pass through the array. Initially
      // lastPos is the index of the last element in the array.
      // During each iteration, it is decreased by one.
      for (lastPos = array.length - 1; lastPos >= 0; lastPos--)
      {
         // The inner loop steps through the array, comparing
         // each element with its neighbor. All of the elements
         // from index 0 through lastPos are involved in the
         // comparison. If two elements are out of order, they
         // are swapped.
         for (index = 0; index <= lastPos - 1; index++)
         {
            // Compare an element with its neighbor.
            if (array[index].compareTo(array[index + 1]) > 0)
            {
               // Swap the two elements.
               temp = array[index];
               array[index] = array[index + 1];
               array[index + 1] = temp;
            }
         }
      }
   }
}
```

**NOTE:** If you compile the class shown in Code Listing 16-3 you will likely see a warning such as:

```
ObjectBubbleSorter.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

For now, ignore this warning. In Chapter 17, when we discuss generic types, you will learn how to eliminate these warnings.

Notice in line 16 that the `bubbleSort` method's parameter is declared as `Comparable[] array`. When we call this method, the argument that we pass must be an array of objects that implement `Comparable`. The comparison that is made in line 36 reads as follows:

```
if (array[index].compareTo(array[index + 1]) > 0)
```

This statement uses the `array[index]` object's `compareTo` method to determine whether it is greater than the object at `array[index + 1]`. If it is, the two elements are swapped. (Also notice that the `temp` variable's data type has been changed. It is now a `Comparable` reference variable.)

Because the `String` class implements the `Comparable` interface, an easy way to test the method is to pass an array of `String` objects to it, as shown in Code Listing 16-4.

### **Code Listing 16-4** (`ObjectBubbleSortTest.java`)

```
/**
   This program tests the bubbleSort method in the
   ObjectBubbleSorter class.
*/

public class ObjectBubbleSortTest
{
   public static void main(String[] args)
   {
      // Create a String array with test values.
      // Note that String implements Comparable.
      String[] values = { "David", "Abe", "Katherine",
                          "Beth", "Jeff", "Daisy" };

      // Display the array's contents.
      System.out.println("Original order:");
      for (String element : values)
         System.out.print(element + " ");

      // Sort the array.
      ObjectBubbleSorter.bubbleSort(values);

      // Display the array's contents.
      System.out.println("\nSorted order:");
      for (String element : values)
         System.out.print(element + " ");

      System.out.println();
   }
}
```

**Program Output**

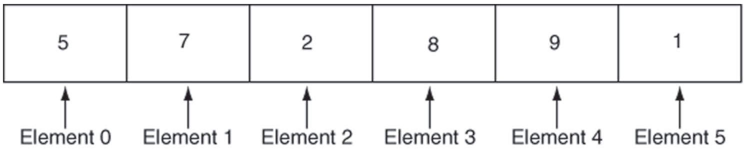
```
Original order:
David Abe Katherine Beth Jeff Daisy
Sorted order:
Abe Beth Daisy David Jeff Katherine
```

When you are designing a class of your own, and you want to be able to sort objects of that class using the `bubbleSort` method shown in the `ObjectBubbleSorter` class (Code Listing 16-3), your class must implement the `Comparable` interface, and provide a `compareTo` method. In Chapter 17, Generics, we will take a closer look at the `Comparable` interface and how to implement it.

## The Selection Sort

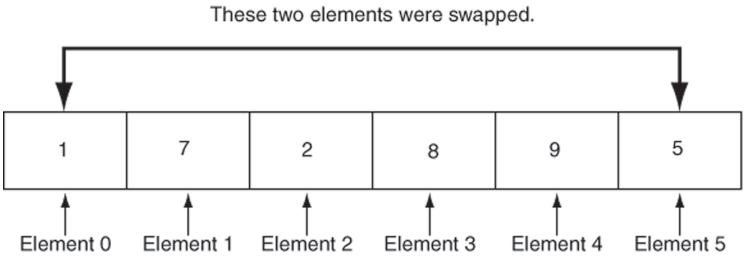
The *selection sort* works like this: The smallest value in the array is located and moved to position 0. Then, the next smallest value is located and moved to position 1. This process continues until all of the elements have been placed in their proper order. Let's see how the selection sort works when arranging the elements of the array in Figure 16-8.

**Figure 16-8** Values in an array



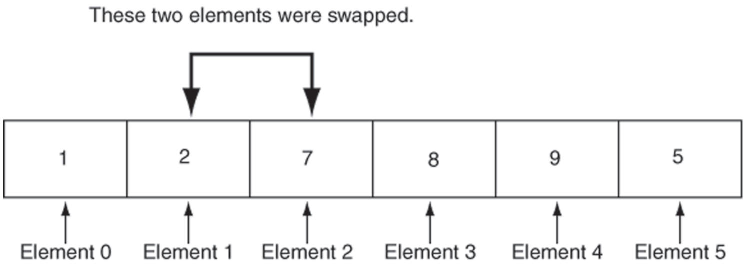
The selection sort scans the array, starting at position 0, and locates the element with the smallest value. Then, this smallest element is swapped with the element at position 0. In this example, the 1 stored at position 5 is swapped with the 5 stored at position 0. After the exchange, the array appears as shown in Figure 16-9.

**Figure 16-9** Values in the array after the first swap



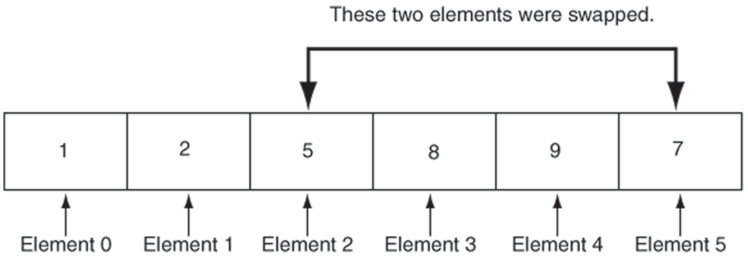
Then, the algorithm repeats the process, but because the element at position 0 is already the smallest value in the array, it can be left out of the procedure. This time, the algorithm begins the scan at position 1. In this example, the element at 2 is exchanged with the element at position 1. Then, the array appears as shown in Figure 16-10.

**Figure 16-10** Values in the array after the second swap


These two elements were swapped.
1
2
7
8
9
5
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

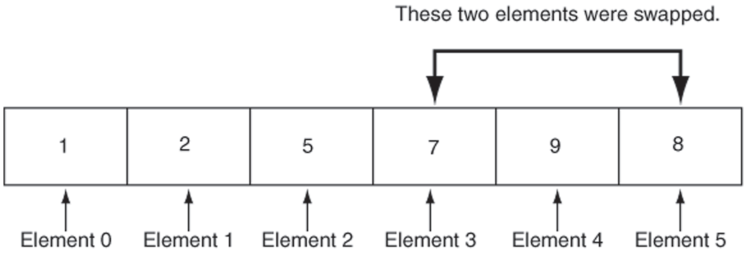
Once again the process is repeated, but this time the scan begins at position 2. The algorithm will find that position 5 contains the next smallest value. This element is swapped with the element at position 2, causing the array to appear as shown in Figure 16-11.

**Figure 16-11** Values in the array after the third swap


These two elements were swapped.
1
2
5
8
9
7
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

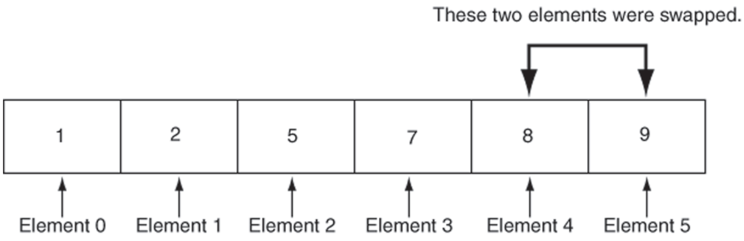
Next, the scanning begins at position 3. Its value is swapped with that at position 5, causing the array to appear as shown in Figure 16-12.

**Figure 16-12** Values in the array after the fourth swap


These two elements were swapped.
1
2
5
7
9
8
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5

At this point, there are only two elements left in the unsorted portion of the array. The algorithm finds that the value at 5 is smaller than that at position 4, so the two are swapped. This puts the array in its final arrangement, as shown in Figure 16-13.

**Figure 16-13** Values in the array after the fifth swap



Here is the selection sort algorithm in pseudocode:

```
For startScan equals each subscript in array from 0 through the next-to-last subscript
   Set index variable to startScan
   Set minIndex variable to startScan
   Set minValue variable to array[startScan]
   For index is each subscript in array from (startScan + 1) through the last subscript
      If array[index] is less than minValue
          Set minValue to array[index]
          Set minIndex to index
      End If
      Increment index
   End For
   Set array[minIndex] to array[startScan]
   Set array[startScan] to minValue
End For
```

Code Listing 16-5 shows the `IntSelectionSorter` class, which has a public static method, `selectionSort`. You pass an `int` array to the method and it uses the selection sort algorithm to sort the array's contents in ascending order. The program in Code Listing 16-6 tests the method.

**Code Listing 16-5** (`IntSelectionSorter.java`)

```
/**
   The IntSelectionSorter class provides a public static
   method for performing a selection sort on an int array.
*/

public class IntSelectionSorter
{

   /**
      The selectionSort method performs a selection sort on an
      int array. The array is sorted in ascending order.
      @param array The array to sort.
   */

   public static void selectionSort(int[] array)
   {
      int startScan; // Starting position of the scan
      int index;     // To hold a subscript value
      int minIndex;  // Element with smallest value in the scan
      int minValue;  // The smallest value found in the scan

      // The outer loop iterates once for each element in the
      // array. The startScan variable marks the position where
      // the scan should begin.
      for (startScan = 0; startScan < (array.length-1); startScan++)
      {
         // Assume the first element in the scannable area
         // is the smallest value.
         minIndex = startScan;
         minValue = array[startScan];

         // Scan the array, starting at the 2nd element in
         // the scannable area. We are looking for the smallest
         // value in the scannable area.
         for(index = startScan + 1; index < array.length; index++)
         {
            if (array[index] < minValue)
            {
               minValue = array[index];
               minIndex = index;
            }
         }

         // Swap the element with the smallest value
         // with the first element in the scannable area.
         array[minIndex] = array[startScan];
         array[startScan] = minValue;
      }
   }
}
```

**Code Listing 16-6** (`SelectionSortTest.java`)

```
/**
   This program tests the selectionSort method in the
   IntSelectionSorter class.
*/
```

```

public class SelectionSortTest
{
   public static void main(String[] args)
   {
      // Create an int array with test values.
      int[] values = { 5, 1, 3, 6, 4, 2 };

      // Display the array's contents.
      System.out.println("Original order:");
      for (int element : values)
         System.out.print(element + " ");

      // Sort the array.
      IntSelectionSorter.selectionSort(values);

      // Display the array's contents.
      System.out.println("\nSorted order:");
      for (int element : values)
         System.out.print(element + " ");

      System.out.println();
   }
}
```

**Program Output**

```
Original order:
5 1 3 6 4 2
Sorted order:
1 2 3 4 5 6
```

## Using the Selection Sort to Sort Objects

Previously, we demonstrated how the `bubbleSort` method could be written to accept an array of `Comparable` objects, and then use the `compareTo` method in the bubble sort algorithm to compare the objects. We can take the same approach with the selection sort, as shown in Code Listing 16-7. This class has a public static `selectionSort` method that accepts an array of `Comparable` objects.

**Code Listing 16-7** (`ObjectSelectionSorter.java`)

```
/**
   The ObjectSelectionSorter class provides a public static
   method for performing a selection sort on an array of
   objects that implement the Comparable interface.
*/
```

```java

public class ObjectSelectionSorter
{

   /**
      The selectionSort method performs a selection sort on an
      array of objects that implement the Comparable interface.
      @param array The array to sort.
   */

   public static void selectionSort(Comparable[] array)
   {
      int startScan;       // Starting position of the scan
      int index;           // To hold a subscript value
      int minIndex;        // Element with smallest value in the scan
      Comparable minValue; // The smallest value found in the scan

      // The outer loop iterates once for each element in the
      // array. The startScan variable marks the position where
      // the scan should begin.
      for (startScan = 0; startScan < (array.length-1); startScan++)
      {
         // Assume the first element in the scannable area
         // is the smallest value.
         minIndex = startScan;
         minValue = array[startScan];

         // Scan the array, starting at the 2nd element in
         // the scannable area. We are looking for the smallest
         // value in the scannable area.
         for(index = startScan + 1; index < array.length; index++)
         {
            if (array[index].compareTo(minValue) < 0)
            {
               minValue = array[index];
               minIndex = index;
            }
         }

         // Swap the element with the smallest value
         // with the first element in the scannable area.
         array[minIndex] = array[startScan];
         array[startScan] = minValue;
      }
   }
}
```

The program in Code Listing 16-8 tests the `selectionSort` method with an array of `String` objects.

**Code Listing 16-8** `(ObjectSelectionSortTest.java)`

```java
/**
   This program tests the selectionSort method in the
   ObjectSelectionSorter class.
*/

public class ObjectSelectionSortTest
{
   public static void main(String[] args)
   {
      // Create a String array with test values.
      // Note that String implements Comparable.
      String[] values = { "David", "Abe", "Katherine",
                          "Beth", "Jeff", "Daisy" };

      // Display the array's contents.
      System.out.println("Original order:");
      for (String element : values)
         System.out.print(element + " ");

      // Sort the array.
      ObjectSelectionSorter.selectionSort(values);

      // Display the array's contents.
      System.out.println("\nSorted order:");
      for (String element : values)
         System.out.print(element + " ");

      System.out.println();
   }
}
```

**Program Output**

```
Original order:
David Abe Katherine Beth Jeff Daisy
Sorted order:
Abe Beth Daisy David Jeff Katherine
```

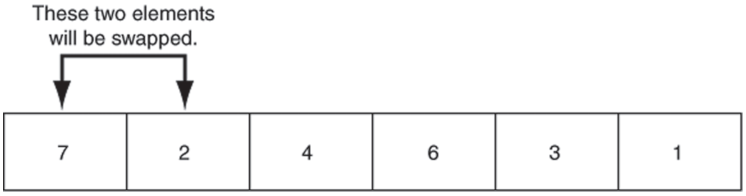
## The Insertion Sort

In the insertion sort, we begin by looking at the first two elements of the array. We compare these elements and, if necessary, we swap them so they are in the proper order. This becomes a sorted portion of the array.

Then, our objective is to incorporate the third element of the array into the part of the array already sorted. We do this by inserting it into proper position, relative to the first two elements. If we need to shift either of the first two elements to accommodate the third element, we do so. Once we have inserted the third element into the correct position (relative to the first two elements), the initial portion of the array consisting of the first three elements will be sorted.

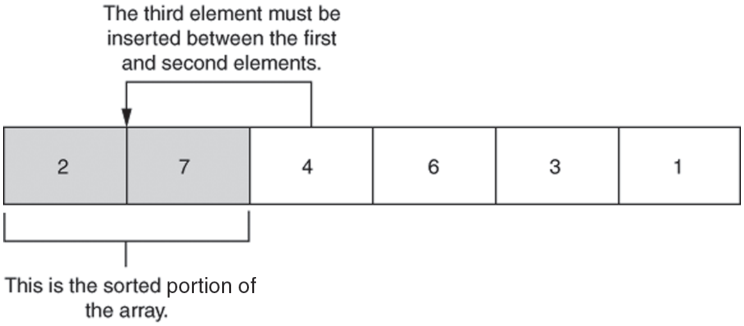
This process continues with the fourth and subsequent elements, until all of the elements have been inserted into their proper positions. Let's look at an example. Suppose we start with the `int` array shown in Figure 16-14. As shown in the figure, the values in the first and second elements are out of order, so they will be swapped.

**Figure 16-14** An unsorted array



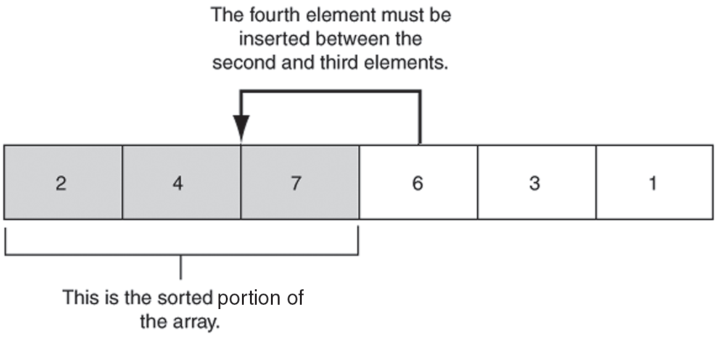
After the swap takes place, the first and second elements will be in sorted order. The next step is to move the value of the third element so it is in the correct position, relative to the first two elements. As shown in Figure 16-15, the value in the third element must be positioned between the values in the first and second elements.

**Figure 16-15** The third element must be moved


The third element must be inserted between the first and second elements.
2
7
4
6
3
1
This is the sorted portion of the array.

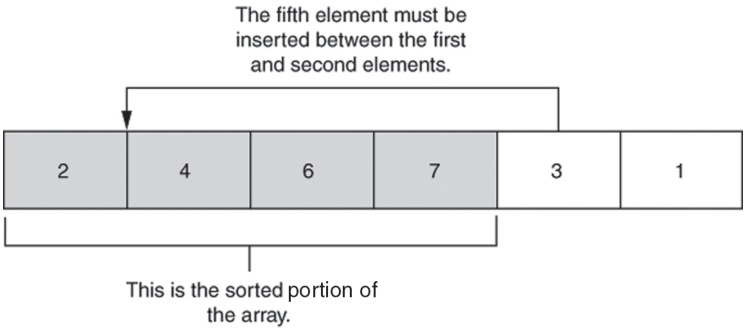
After the value in the third element is moved to its new position, the first three elements will be in sorted order. The next step is to move the value of the fourth element so it is in the correct position, relative to the first three elements. As shown in Figure 16-16, the value in the fourth element must be positioned between the values in the second and third elements.

**Figure 16-16** The fourth element must be moved


The fourth element must be inserted between the second and third elements.
2
4
7
6
3
1
This is the sorted portion of the array.

After the value in the fourth element is moved to its new position, the first four elements will be in sorted order. The next step is to move the value of the fifth element so it is in the correct position, relative to the first four elements. As shown in Figure 16-17, the value in the fifth element must be positioned between the values in the first and second elements.

**Figure 16-17** The fifth element must be moved


The fifth element must be inserted between the first and second elements.
2
4
6
7
3
1
This is the sorted portion of the array.

After the value in the fifth element is moved to its new position, the first five elements will be in sorted order. The next step is to move the value of the sixth element so it is in the correct position, relative to the first five elements. As shown in Figure 16-18, the value in the sixth element must be moved to the beginning of the array.

The sixth element is the last element in the array. Once it is moved to its correct position, the entire array is sorted. This is shown in Figure 16-19.

**Figure 16-18** The sixth element must be moved

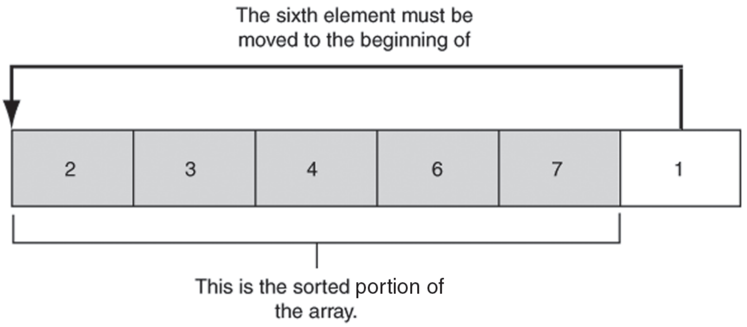
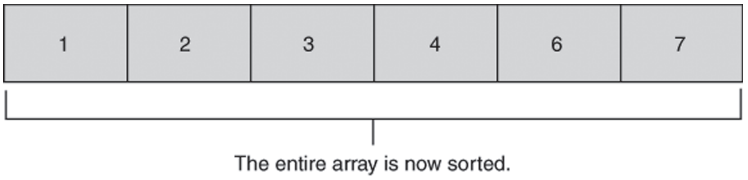

The sixth element must be
moved to the beginning of
2
3
4
6
7
1
This is the sorted portion of
the array.

**Figure 16-19** All of the elements are in the correct position


1
2
3
4
6
7
The entire array is now sorted.

Here is the insertion sort algorithm in pseudocode:

```
For index equals each subscript in array from 1 through the last subscript
    Save the value in index in scan
    Save the value in array[index]in unsortedValue
    While scan is greater than 0 and array[scan – 1] is greater than unsortedValue
        Move array[scan – 1] to array[scan]
        Subtract 1 from scan
    End While
    Store the value in unsortedValue in array[scan]
End For
```

Code Listing 16-9 shows the `IntInsertionSorter` class, which has a public static method, `insertionSort`. You pass an `int` array to the method and it uses the insertion sort algorithm to sort the array's contents in ascending order. The program in Code Listing 16-10 tests the method.

**Code Listing 16-9** (`IntInsertionSorter.java`)

```
/**
   The IntInsertionSorter class provides a public static
   method for performing an insertion sort on an int array.
*/
```

```

public class IntInsertionSorter
{

   /**
      The insertionSort method performs an insertion sort on
      an int array. The array is sorted in ascending order.
      @param array The array to sort.
   */

   public static void insertionSort(int[] array)
   {
      int unsortedValue;  // The first unsorted value
      int scan;           // Used to scan the array

      // The outer loop steps the index variable through each
      // subscript in the array, starting at 1. The portion of the
      // array consisting of element 0 by itself is already sorted.
      for (int index = 1; index < array.length; index++)
      {
         // The first element outside the sorted portion is
         // array[index]. Store the value of this element
         // in unsortedValue.
         unsortedValue = array[index];

         // Start scan at the subscript of the first element
         // in the still unsorted part.
         scan = index;

         // Move the first element in the still unsorted part
         // into its proper position within the sorted part.
         while (scan > 0 && array[scan – 1] > unsortedValue)
         {
            array[scan] = array[scan-1];
            scan--;
         }

         // Insert the unsorted value in its proper position
         // within the sorted part.
         array[scan] = unsortedValue;
      }
   }
}
```

**Code Listing 16-10** `(InsertionSortTest.java)`

```
/**
   This program tests the insertionSort method in the
   IntInsertionSorter class.
*/

public class InsertionSortTest
{
   public static void main(String[] args)
   {
      // Create an int array with test values.
      int[] values = { 5, 1, 3, 6, 4, 2 };

      // Display the array's contents.
      System.out.println("Original order:");
      for (int element : values)
         System.out.print(element + " ");

      // Sort the array.
      IntInsertionSorter.insertionSort(values);

      // Display the array's contents.
      System.out.println("\nSorted order:");
      for (int element : values)
         System.out.print(element + " ");

      System.out.println();
   }
}
```

**Program Output**

```
Original order:
5 1 3 6 4 2
Sorted order:
1 2 3 4 5 6
```

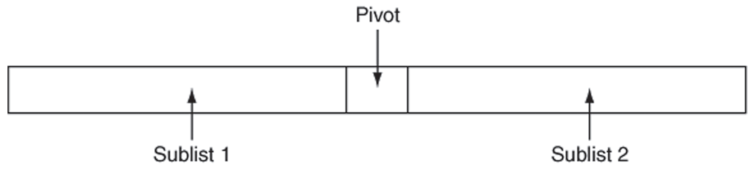
## Using the Insertion Sort to Sort Objects

We have demonstrated how the `bubbleSort` and `selectionSort` methods can be written to accept an array of `Comparable` objects, and then use the `compareTo` method to compare the objects. A similar approach can be taken with the `insertionSort` method. This will be left as a programming exercise for you at the end of the chapter.

## The Quicksort Algorithm

The Quicksort algorithm is a popular sorting routine developed in 1960 by C. A. R. Hoare. It sorts an array by dividing it into two sublists. Between the sublists is a selected value known as the *pivot*, as shown in Figure 16-20.

**Figure 16-20** Sublists and pivot



Notice in the figure that sublist 1 is positioned to the left of (before) the pivot, and sublist 2 is positioned to the right of (after) the pivot. Once a pivot value has been selected, the algorithm reorders the other values in the array until all the elements in the left sublist are less than the pivot, and all the elements in the right sublist are greater than or equal to the pivot. This does not mean that the elements in the sublists are sorted in order, just that the smaller values in the array are in the left sublist and the larger values are in the right sublist.

Once this is done, the algorithm recursively repeats the procedure on sublist 1, and then on sublist 2. During this process, the sublists are themselves divided into sublists with a pivot. All the elements less than the pivot are moved to the left sublist, and all the elements greater than the pivot are moved to the right sublist. The recursion stops when there is at most one element in a sublist. When all of the recursive calls have completed, the array is completely sorted.

To demonstrate the Quicksort algorithm we have provided a class, `IntQuickSorter.java`. This class has the following public static method:

```
public static void quickSort(int array[])
{
   doQuickSort(array, 0, array.length - 1);
}
```

This method accepts an `int` array as its argument. Then it calls a private static method, `doQuickSort`. The reason we have this extra method call is that the Quicksort algorithm requires the starting subscript and the ending subscript of the area of the array that is to be sorted. When you first invoke the Quicksort algorithm, you provide the starting and ending subscripts of the entire array. As a convenience to anyone using the `IntQuickSorter` class, we have provided the `quickSort` method, which simply accepts an array. Then it calls the `doQuickSort` method, providing the starting and ending subscripts as arguments.

The pseudocode for the `doQuickSort` method is shown here:

```
doQuickSort:
  If Starting Index < Ending Index
      Partition the List around a Pivot
      doQuickSort Sublist 1
      doQuickSort Sublist 2
  End If
End doQuickSort
```

Here is the Java code for the `doQuickSort` method:

```
27   private static void doQuickSort(int array[], int start, int end)
28   {
29      int pivotPoint;
30
31      if (start < end)
32      {
33         // Get the pivot point.
34         pivotPoint = partition(array, start, end);
35
36         // Sort the first sublist.
37         doQuickSort(array, start, pivotPoint − 1);
38
39         // Sort the second sublist.
40         doQuickSort(array, pivotPoint + 1, end);
41      }
42   }
```

The first argument is the array holding the list that is to be sorted. The second and third arguments are the starting and ending subscripts of the list.

The subscript of the pivot element is returned by the `partition` method in line 34. The `partition` method not only determines which element will be the pivot, but also rearranges the other values in the list. The method selects the element in the middle of the array as the pivot, and then scans the remainder of the array searching for values less than the pivot.

The code for the `partition` method is shown here:

```
56   private static int partition(int array[], int start, int end)
57   {
58      int pivotValue;      // To hold the pivot value
59      int endOfLeftList;   // Last element in the left sublist.
60      int mid;             // To hold the mid-point subscript
61
62      // Find the subscript of the middle element.
63      // This will be our pivot value.
64      mid = (start + end) / 2;
65
```

```
      // Swap the middle element with the first element.
      // This moves the pivot value to the start of
      // the list.
      swap(array, start, mid);

      // Save the pivot value for comparisons.
      pivotValue = array[start];

      // For now, the end of the left sublist is
      // the first element.
      endOfLeftList = start;

      // Scan the entire list and move any values that
      // are less than the pivot value to the left
      // sublist.
      for (int scan = start + 1; scan <= end; scan++)
      {
         if (array[scan] < pivotValue)
         {
            endOfLeftList++;
            swap(array, endOfLeftList, scan);
         }
      }

      // Move the pivot value to end of the
      // left sublist.
      swap(array, start, endOfLeftList);

      // Return the subscript of the pivot value.
      return endOfLeftList;
   }
```

The `partition` method does not initially sort the values into their final order. Its job is only to move the values that are less than the pivot to the pivot's left, and move the values that are greater than the pivot to the pivot's right. As long as that condition is met, they may appear in any order. The ultimate sorting order of the entire array is achieved cumulatively, through the recursive calls to the `doQuickSort` method.

There are many different ways of partitioning the array. As previously stated, the technique shown in this `partition` method selects the middle value as the pivot. That value is then moved to the beginning of the array (by swapping it with the value stored there). This simplifies the next step, which is to scan the array.

A `for` loop scans the remainder of the array, and when an element is found whose value is less than the pivot, that value is moved to a location left of the pivot point. A third method, `swap`, is used to swap the values found in any two elements of the array. The method is shown here:

```java
private static void swap(int[] array, int a, int b)
{
   int temp;

   temp = array[a];
   array[a] = array[b];
   array[b] = temp;
}
```

The program shown in Code Listing 16-11 uses these methods to test the `Quicksort` algorithm.

**Code Listing 16-11** (`QuickSortTest.java`)

```java
/**
   This program tests the quickSort method in the
   IntQuickSorter class.
*/

public class QuickSortTest
{
   public static void main(String[] args)
   {
      // Create an int array with test values.
      int[] values = { 5, 1, 3, 6, 4, 2 };

      // Display the array's contents.
      System.out.println("Original order:");
      for (int element : values)
         System.out.print(element + " ");

      // Sort the array.
      IntQuickSorter.quickSort(values);

      // Display the array's contents.
      System.out.println("\nSorted order:");
      for (int element : values)
         System.out.print(element + " ");

      System.out.println();
   }
}
```

**Program Output**

```
Original order:
5 1 3 6 4 2
Sorted order:
1 2 3 4 5 6
```

### Using Quicksort to Sort Objects

As with the other sorting algorithms shown in this chapter, the Quicksort methods we have demonstrated can be modified to accept an array of `Comparable` objects, and then we can use the `compareTo` method to make comparisons. Modifying these Quicksort methods to accept an array of `Comparable` objects will be left as a programming exercise for you at the end of the chapter.

**Checkpoint**

16.1 Which of the sorting algorithms that we discussed makes several passes through an array and causes the larger values to move gradually toward the end of the array with each pass?

16.2 One of the sorting algorithms that we discussed works like this: It begins by putting the initial portion of the array consisting of the first two elements in sorted order. Then the third element is moved to its correct position, relative to the first two elements. At that point the first three elements are in sorted order. This process continues with the fourth and subsequent elements until the entire array is sorted. Which algorithm is this?

16.3 One of the sorting algorithms that we discussed works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order. Which algorithm is this?

16.4 One of the algorithms we discussed divides an array into two sublists, with a pivot value between them. It arranges the sublists so that all the values less than the pivot are stored in the left sublist and all the values greater than the pivot are stored in the right sublist. The algorithm recursively sorts the sublists in this same way. When all of the recursive calls have completed, the array is completely sorted. Which algorithm is this?

## 16.2 Introduction to Search Algorithms

**CONCEPT: A search algorithm is a method of locating a specific item in a larger collection of data. This section discusses two algorithms for searching the contents of an array.**

It's very common for programs not only to store and process data stored in arrays, but also to search arrays for specific items. This section will show you two methods of searching an array: the sequential search and the binary search. Each has its advantages and disadvantages.

### The Sequential Search Algorithm

The *sequential search algorithm* uses a loop to step sequentially through an array, starting with the first element. It compares each element with the value being searched for and stops when the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm unsuccessfully searches to the end of the array.

Here is the pseudocode for a method that performs the sequential search:

```
Set found to false
Set position to -1
Set index to 0
While found is false and index < number of elements
    If list[index] is equal to search value
        found = true
        position = index
    End If
    Add 1 to index
End While
Return position
```

This pseudocode method returns the subscript of the element holding the value. If the value is not found, however, it returns −1. The reason −1 is returned when the value is not found is that −1 is not a valid array subscript, and it cannot be mistaken as a position within the array.

The `IntSequentialSearcher` class in Code Listing 16-12 has a public static method that performs a sequential search on an `int` array. The program in Code Listing 16-13 tests the method.

**Code Listing 16-12** (`IntSequentialSearcher.java`)

```
/**
   The IntSequentialSearcher class provides a public static
   method for performing a sequential search on an int array.
*/

public class IntSequentialSearcher
{
   /**
      The search method searches an array for a value.
      @param array The array to search.
      @param value The value to search for.
      @return The subscript of the value if found in the
              array, otherwise -1.
   */

   public static int search(int[] array, int value)
   {
      int index;     // Loop control variable
      int position;  // Position the value is found at
      boolean found; // Flag indicating search results

      // Element 0 is the starting point of the search.
      index = 0;

      // Store the default values position and found.
      position = -1;
      found = false;

      // Search the array.
      while (!found && index < array.length)
      {
         if (array[index] == value)
         {
            found = true;
            position = index;
         }
         index++;
      }

      // Return the found element's position,
      // or -1 if not found.
      return position;
   }
}
```

**Code Listing 16-13** (`SequentialSearchTest.java`)

```
/**
   This program sequentially searches an
   int array for a specified value.
*/

public class SequentialSearchTest
{
   public static void main(String[] args)
   {
      int[] tests = { 87, 75, 98, 100, 82 };
      int results;

      // Search the array for the value 100.
      results = IntSequentialSearcher.search(tests, 100);

      // Determine whether 100 was found and
      // display an appropriate message.
      if (results == -1)
      {
         System.out.println("You did not " +
                            "earn 100 on any test.");
      }
      else
      {
         System.out.println("You earned 100 " +
                            "on test " + (results + 1));
      }
   }
}
```

**Program Output**

```
You earned 100 on test 4
```

## Average Case Analysis of the Sequential Search

The advantage of the sequential search is its simplicity. It is very easy to understand and implement. Sequential search is very flexible, and can be used on any array. But this flexibility comes at a cost: If the array being searched contains 20,000 elements, the algorithm may have to look at all 20,000 elements. This might happen, for example, when the value being searched for is not in the array, or when it is found in the very last array entry.

In an average case, an item is just as likely to be found near the beginning of the array as near the end. Typically, for an array of $n$ items, the sequential search will locate an item in $n/2$ attempts. If an array has 50,000 elements, the sequential search will make a comparison with 25,000 of them in a typical case. This is assuming, of course, that the search item has equal probability of being found at any position in the array. ($n/2$ is the average number of comparisons. The maximum number of comparisons is always $n$.)

## The Binary Search

The *binary search* is a clever algorithm that is much more efficient than the sequential search. Its only requirement is that the values in the array must be sorted in some order. Let us suppose the input array is sorted in ascending order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value isn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm tests *its* middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found, or there are no more elements to test. Here is the pseudocode for a method that performs a binary search on an array:

```
Set first to 0
Set last to the last subscript in the array
Set found to false
Set position to –1
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last]
    If array[middle] equals the desired value
        Set found to true
        Set position to middle
    Else If array[middle] is greater than the desired value
        Set last to middle – 1
    Else
        Set first to middle + 1
    End If
End While
Return position
```

This algorithm uses three variables to mark positions within the array: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's `first` and `last` elements. The subscript of the element halfway between `first` and `last` is calculated and stored in the `middle` variable. If the element in the middle of the array does not contain the search value, the `first` or `last` variables are adjusted so that only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The `IntBinarySearcher` class shown in Code Listing 16-14 has a public static method that performs a binary search on an integer array. The first parameter, `array`, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, −1 is returned indicating the value did not appear in the array. The program in Code Listing 16-15 tests the method.

**Code Listing 16-14** (IntBinarySearcher.java)

```java
/**
   The IntBinarySearcher class provides a public static
   method for performing a binary search on an int array.
*/

public class IntBinarySearcher
{
   /**
      The search method performs a binary search on an int
      array. The array is searched for the number passed to
      value. If the number is found, its array subscript is
      returned. Otherwise, –1 is returned indicating the
      value was not found in the array.
      @param array The array to search.
      @param value The value to search for.
   */

   public static int search(int[] array, int value)
   {
      int first;     // First array element
      int last;      // Last array element
      int middle;    // Mid point of search
      int position;  // Position of search value
      boolean found; // Flag

      // Set the initial values.
      first = 0;
      last = array.length – 1;
      position = –1;
      found = false;

      // Search for the value.
      while (!found && first <= last)
      {
         // Calculate midpoint
         middle = (first + last) / 2;

         // If value is found at mid point...
         if (array[middle] == value)
         {
            found = true;
            position = middle;
         }
         // else if value is in lower half...
         else if (array[middle] > value)
            last = middle – 1;
         // else if value is in upper half....
         else
            first = middle + 1;
      }

      // Return the position of the item, or –1
      // if it was not found.
      return position;
   }
}
```

**Code Listing 16-15** (`BinarySearchTest.java`)

```
import java.util.Scanner;

/**
   This program demonstrates the search method in
   the IntBinarySearcher class.
*/

public class BinarySearchTest
{
   public static void main(String[] args)
   {
      int result, searchValue;
      String input;

      // An array of numbers to search.
      int numbers[] = { 536, 289, 296, 429, 319, 142, 394,
                        101, 388, 147, 417, 199, 207, 222,
                        189, 310, 447, 521, 234, 600};

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // First we must sort the array in ascending order.
      IntQuickSorter.quickSort(numbers);

      do
      {
         // Get a value to search for.
         System.out.print("Enter a value to search for: ");
         searchValue = keyboard.nextInt();

         // Search for the value
         result = IntBinarySearcher.search(numbers, searchValue);

         // Display the results.
         if (result == -1)
           System.out.println(searchValue + " was not found.");
         else
         {
            System.out.println(searchValue + " was found at " +
                               "element " + result);
         }

         // Consume the remaining newline...
         keyboard.nextLine();

         // Does the user want to search again?
         System.out.print("Do you want to search again? (Y or N): ");
         input = keyboard.nextLine();
      } while (input.charAt(0) == 'y' || input.charAt(0) == 'Y');
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter a value to search for: 289 [Enter]
289 was found at element 8
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 101 [Enter]
101 was found at element 0
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 999 [Enter]
999 was not found.
Do you want to search again? (Y or N): n [Enter]
```

## Efficiency of the Binary Search

For sorted arrays, the binary search is much more efficient than the sequential search. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched. For example, consider an array with 1,000 elements.

If the binary search fails to find an item on the first attempt, the number of elements that remains to be searched is 500. If the item is not found on the second attempt, the number of elements that remains to be searched is 250. This process continues until the binary search has either located the desired item or determined that it is not in the array. With 1,000 elements this takes no more than 10 comparisons. (Compare this to the sequential search, which would make an average of 500 comparisons!)

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array of any size. Simply find the smallest power of 2 that is greater than or equal to the number of elements in the array. For example, a maximum of 16 comparisons will be made on an array of 50,000 elements ($2^{16} = 65{,}536$), and a maximum of 20 comparisons will be made on an array of 1,000,000 elements ($2^{20} = 1{,}048{,}576$).

## A Recursive Binary Search

The binary search algorithm that we previously examined used an iterative approach, meaning that it used a loop to step through the array. The binary search algorithm can also be implemented recursively. For example, the procedure can be expressed as follows:

```
If array[middle] equals the search value, then the value is found
Else if array[middle] is less than the search value, perform a binary
        search on the upper half of the array
Else if array[middle] is greater than the search value, perform
        a binary search on the lower half of the array
End If
```

When you compare the recursive algorithm to its iterative counterpart, it becomes evident that the recursive version is much more elegant and easier to understand. The recursive binary search algorithm is also a good example of repeatedly breaking a problem down into smaller pieces until it is solved. Code Listing 16-16 shows the `RecursiveBinarySearcher` class. This class provides a public static method named `search`, which calls the recursive `binarySearch` method to find a value in an `int` array. The program in Code Listing 16-17 tests the method.

**Code Listing 16-16** (`RecursiveBinarySearcher.java`)

```java
/**
   The RecursiveBinarySearcher class provides a public static
   method for performing a recursive binary search on an int array.
*/

public class RecursiveBinarySearcher
{
   /**
      The search method calls the doBinarySearch method
      to search for a value in an array.
      @param array The array to search.
      @param value The value to search for.
   */

   public static int search(int[] array, int value)
   {
      return binarySearch(array, 0, array.length - 1, value);
   }

   /**
      The binarySearch method performs a recursive binary
      search on an integer array.
      @param array The array to search.
      @param first The first element in the search range.
      @param last The last element in the search range.
      @param value The value to search for.
      @return The subscript of the value if found,
              otherwise -1.
   */

   private static int binarySearch(int[] array, int first,
                                   int last, int value)
   {
      int middle;     // Mid point of search

      // Test for the base case where the
      // value is not found.
      if (first > last)
         return -1;

      // Calculate the middle position.
      middle = (first + last) / 2;

      // Search for the value.
      if (array[middle] == value)
         return middle;
      else if (array[middle] < value)
         return binarySearch(array, middle + 1,
                             last, value);
      else
         return binarySearch(array, first,
                             middle - 1, value);
   }

}
```

**Code Listing 16-17** (RecursiveBinarySearchTest.java)

```
import java.util.Scanner;

/**
   This program tests the search method in the
   RecursiveBinarySearcher class.
*/

public class RecursiveBinarySearchTest
{
   public static void main(String [] args)
   {
      int result, searchValue;
      String input;

      // An array of numbers to search.
      int numbers[] = { 536, 289, 296, 429, 319, 142, 394,
                        101, 388, 147, 417, 199, 207, 222,
                        189, 310, 447, 521, 234, 600};

      // Create a Scanner object to read keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // First we must sort the array in ascending order.
      IntQuickSorter.quickSort(numbers);

      do
      {
         // Get a value to search for.
         System.out.print("Enter a value to search for: ");
         searchValue = keyboard.nextInt();

         // Search for the value
         result =
            RecursiveBinarySearcher.search(numbers, searchValue);

         // Display the results.
         if (result == -1)
            System.out.println(searchValue + " was not found.");
         else
         {
            System.out.println(searchValue + " was found at " +
                               "element " + result);
         }

         // Consume the remaining newline.
         keyboard.nextLine();

         // Does the user want to search again?
         System.out.print("Do you want to search again? (Y or N): ");
         input = keyboard.nextLine();
      } while (input.charAt(0) == 'y' || input.charAt(0) == 'Y');
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter a value to search for: 289 [Enter]
289 was found at element 8
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 101 [Enter]
101 was found at element 0
Do you want to search again? (Y or N): y [Enter]
Enter a value to search for: 999 [Enter]
999 was not found.
Do you want to search again? (Y or N): n [Enter]
```

**Checkpoint**

16.5 Describe the difference between the sequential search and the binary search.

16.6 On average, with an array of 20,000 elements, how many comparisons will the sequential search perform? (Assume the items being searched have equal probability of being found at any of the positions in the array.)

16.7 With an array of 20,000 elements, what is the maximum number of comparisons the binary search will perform?

16.8 If a sequential search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

## 16.3 Analysis of Algorithms

**CONCEPT: We can estimate the efficiency of an algorithm by counting the number of steps that the algorithm requires to solve a problem.**

An algorithm is a mechanical step-by-step procedure for solving a problem and is the basic strategy used in designing a program. There is often more than one algorithm that can be used to solve a given problem. For example, we saw in the previous section that the problem of searching a sorted array can be solved by two different methods: *sequential search* and *binary search*.

How can we decide which of two algorithms for solving a problem is better? To answer this question, we need to establish criteria for judging the "goodness" or efficiency of an algorithm. The two criteria that are most often used are space and time. The *space* criterion refers to the amount of memory the algorithm requires to solve the problem, while the *time* criterion refers to the length of execution time. In this chapter, we will use the time criterion to evaluate the efficiency of algorithms.

One possibility for comparing two algorithms is to code them and then time the execution of the resulting Java programs. This experimental approach can yield useful information, but it has the following shortcomings:

- It measures the efficiency of programs rather than algorithms.
- The results obtained depend on the programming language used to code the algorithms, and on the quality of the compiler used to generate machine code. The programs may run faster or slower if they are coded in a different language, or compiled by a different compiler.
- The results obtained depend on how the operating system executes programs, and on the nature of the hardware on which the programs are executing. The execution times may be different if we run the programs on a different computer and a different operating system.
- The results obtained apply only to those inputs that were part of the execution runs and may not be representative of the performance of the algorithms on a different set of inputs.

A better approach is to count the number of basic steps an algorithm requires to process an input of a given size. To make sense of this approach, we need more precise definitions of the concepts of a computational problem, problem input, input size, and basic step.

## Computational Problems and Basic Steps

A *computational problem* is a problem to be solved using an algorithm. Such a problem is a collection of *instances*, with each instance being specified by input data given in some prescribed format. For example, if the problem $P$ is to sort an array of integers, then an instance of $P$ is just a specific integer array. The *size* of an instance refers to the amount of memory needed to hold the input data. The input size is usually given as a number that allows us to infer the total number of bits occupied by the input data. If the number of bits occupied by an entry of the array is fixed, say at 64 bits, then the length of the array is a good measure of input size. In contrast, the length of the array is not a good measure of input size if there is no a priori bound on the size of array elements.

A step executed by an algorithm is a *basic step* (also called a *basic operation*) if the algorithm can execute the step in time bounded by a constant regardless of the size of the input. In sorting an array of integers, the step

*Swap the elements in positions* $k$ *and* $k+1$

is basic because the time required to swap two array elements remains constant even if the length of the array increases. In contrast, a step such as

*Find the largest element of the array*

is not basic because the time required to complete the step depends on the length of the array. Intuitively, a basic step is one that can conceivably be built into the hardware of some physical computers.

The definition of a basic step does not specify the size of the constant that bounds the time required to execute the step. Ignoring the exact value of these constants reflects the reality that the same operation may be executed with different speeds on different hardware, and that an operation that can be executed with one hardware instruction on one computer may require several hardware instructions on another computer. A consequence of this definition is that we can count any constant number of basic steps as one basic step. In particular, an algorithm that executes $500n$ basic steps can accurately be described as executing $n$ basic steps.

It is important to realize that ordinary arithmetic and logic operations such as addition and comparison are not basic unless a constant bound is put on the size of the numbers being added or compared. The size of the bound does not matter as long as the bound is constant: it may be 32, 64, 128, 1024 bits, or even larger, and these operations will still be basic. In the following discussion, we assume that numbers that occur in our algorithms as inputs, outputs, or are computed as intermediate results are all bounded in size. This allows us to consider operations on them as basic.

It only makes sense to describe an algorithm after we have described the problem the algorithm is supposed to solve. A computational problem is described by giving, for each positive integer $n$, the format of a typical input of size $n$ followed by a description of output to be produced by an algorithm that solves the problem. For example, the problem of summing an integer array can be described as follows:

INPUT: An integer array $a[\ ]$ of size $n$
SIZE OF INPUT: The number $n$ of array entries
OUTPUT: An integer *sum* representing the sum total of the array entries

In this context, an algorithm for solving the problem can be given as follows:

```
ALGORITHM:
1:   sum = 0
2:   k = 0      //array index
3:   While k < n do
4:      sum = sum + a[k]
5:      k = k + 1
6:   End While
```

## Complexity of Algorithms

We can determine the number of basic steps executed by this algorithm on an input of size $n$ by reasoning as follows. The algorithm consists of two statements (Statements 1 and 2) and a loop (Statements 3–6). The two statements before the loop can be counted as one basic operation. Clearly, the amount of time the algorithm takes to execute a *single* iteration of the loop does not depend on $n$, and so is constant. (This can also be seen by noting that each statement inside the loop is a basic operation.) This allows us to count each loop iteration as a single basic operation. Because the algorithm executes the loop $n$ times, it executes $n + 1$ basic operations altogether. For large problem sizes, the number of basic operations performed is approximately $n$. Because of the hidden constant factors, we say that the algorithm requires time proportional to $n$ to process an input of size $n$.

Notice that in our analysis of the array-summing algorithm, we could have gotten the same result by counting the number of loop iterations performed by the algorithm. Moreover, notice that the crucial operation in summing an array is the addition of array elements (Statement 4), and that there are as many additions of array elements as there are loop iterations. Thus, we could have gotten the same result by just counting additions of array elements. It turns out that for most algorithms, it suffices to identify and count only one or two basic operations that are in some way crucial to the problem being solved. For example, in many array searching and sorting algorithms, it suffices to count the number of comparisons between array elements.

The array-summing algorithm just considered is particularly simple to analyze because it performs the same amount of work for all inputs of a given size. Consider the problem of searching an array of integers for a given integer $X$, solved here by the familiar sequential search algorithm:

INPUT: An integer array $a[\ ]$ of size $n$, and an integer $X$
SIZE OF INPUT: The number $n$ of array entries
OUTPUT: An integer $k$ in the range $0 \leq k \leq n - 1$ such that $a[k] = X$, or $k = n$

```
ALGORITHM:
k = 0
While k < n and a[k] != X do
  k = k + 1;
End While
```

This algorithm starts at one end and searches sequentially through the array. The algorithm stops as soon as it encounters $X$, but will search the entire array if $X$ is not in the array. The algorithm may stop after making only one comparison ($X$ is found in the first entry examined), or it may stop after making $n$ comparisons ($X$ is found in the last place examined, or is not in the array). In fact, the algorithm may perform $m$ comparisons for any value of $m$ between 1 and $n$. In cases where the algorithm may perform different amounts of work for different inputs of the same size, it is common to measure the efficiency of the algorithm by the work done on an input of size $n$ that requires the most work. This is called measuring the algorithm by its *worst-case complexity function*.

## Worst Case Complexity of Algorithms

The worst-case complexity function $f(n)$ of an algorithm is the number of steps it performs on an input of size $n$ that requires the most work. It gives an indication of the longest time an algorithm will ever take to solve an instance of size $n$, and is a good measure of efficiency to use when we are looking for a performance guarantee.

Let us determine the worst-case complexity of binary search. This algorithm is used to locate an item $X$ in an array sorted in ascending order. The worst case occurs when $X$ is not found in the array. We will show that in this case, the algorithm performs $L + 1$ steps, where $L$ is the number of loop iterations.

As shown earlier in this chapter, this algorithm consists of some initialization of some variables followed by a loop. The initialization requires constant time, and can therefore be considered to be one basic operation. Likewise, each iteration of the loop is a basic step (increasing the number of entries in the array does not increase the amount of time required by a single iteration of the loop). This shows that the number of steps required by binary search is $L + 1$. Now $L$ is approximately equal to the integer part of $\log n$, the logarithm of $n$ to the base 2. To see this, notice that the size of the array to be searched is initially $n$, and each iteration reduces the size of the remaining portion of the array by one half. Because each loop iteration performs at most two comparisons, binary search performs a total of $2 \log n$ comparisons. By ignoring the constant factor, we can summarize our findings as follows:

*In the worst case, binary search requires time proportional to log n.*

For our last example of determining the complexity of an algorithm, consider the following outline of a variant of the selection sort algorithm, also studied earlier in this chapter:

INPUT: An array $a[\ ]$ of $n$ integers
SIZE OF INPUT: The number $n$ of array entries
OUTPUT: The array $a[\ ]$ rearranged so that $a[0] \leq a[1] \leq \cdots \leq a[n-1]$

```
ALGORITHM:
For (k = n–1; k >= 1; k--)
    // a[0..k] is what remains to be sorted
    determine position p of largest entry in a[0..k]
    swap a[p] with a[k]
End For
```

Let us determine the number of comparisons (involving array entries) executed by this algorithm in sorting an array of $n$ entries. All such comparisons occur in step 3, which is clearly not a basic step (it requires time proportional to $k$, which can be as large as $n$). Let us express step 3 using operations that are clearly basic:

INPUT: array $a[0..k]$ of $k + 1$ entries
SIZE OF INPUT: number $k + 1$ of array entries

```
 6:  p = 0   //position of largest
 7:  For (m = 1; m <= k; m++)
 8:     If a[m] > a[p] Then
 9:        p = m
10:     End if
11:  End For
```

It is clear that this algorithm requires $k$ comparisons between array entries.

Now returning to the main sorting algorithm, we observe that there will be $n - 1$ iterations of the loop that starts at line 1 and ends at line 5, with one such iteration for each value of $k$ in the range $n - 1$ to 1. Moreover, on the $k$-th such iteration, the step on line 3 will perform $k$ comparisons:

$k = n - 1$: step 3 performs $n - 1$ comparisons
$k = n - 2$: step 3 performs $n - 2$ comparisons
.
.
.
$k = 1$: step 3 performs 1 comparison

Now we see that the total number of comparisons performed by this simple sorting algorithm is given by the expression

$$1 + 2 + 3 + \cdots + n - 1 = (n - 1)n/2$$

For large $n$, this expression is very close to $n^2 / 2$. We have proved the following:

*In the worst case, selection sort requires time proportional to* $n^2$.

## Average Case Complexity

The worst-case complexity will not give a good indication of how an algorithm will perform in practical situations in which inputs that yield worst case performance are rare. An alternative method, the *average case complexity function*, can be used if the relative frequencies with which the different inputs are likely to occur in practice are known. The average case complexity function uses such frequencies to form a weighted average of the number of steps performed on each input. Although it yields a good measure of the expected performance of an algorithm, it is difficult to use in practice because reliable estimates of input frequencies are usually not available.

## Asymptotic Complexity and the Big O Notation

We can compare two algorithms $F$ and $G$ for solving a problem by comparing their complexity functions. More specifically, if $f(n)$ and $g(n)$ are the complexity functions for the two algorithms, we can compare the algorithms against each other by looking at what happens to the ratio $f(n)/g(n)$ when $n$ gets large. This is easiest to understand if this ratio tends to some limit. Let us consider some specific examples. Throughout, we assume that $f(n) \geq 1$ and $g(n) \geq 1$ for all $(n) \geq 1$.

- $f(n) = 3n^2 + 5n$ and $g(n) = n^2$. In this case

  $$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n^2} = 3 + \frac{5}{n} \longrightarrow 3 \text{ as } n \longrightarrow \infty$$

  That is, the value of $f(n)/g(n)$ gets closer and closer to 3 as $n$ gets large. What this means is that for very large input sizes $F$ performs three times as many basic operations as $G$. Because the two algorithms differ in performance by a constant factor, we consider them to be equivalent in efficiency.

- $f(n) = 3n^2 + 5n$ and $g(n) = n$. In this case

  $$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n} = 3n + 5 \longrightarrow \infty \text{ as } n \longrightarrow \infty$$

  Here, the ratio $f(n)/g(n)$ gets larger and larger as $n$ gets large. This means $F$ does a lot more work than $G$ on large input sizes. This makes $G$ the better algorithm for large inputs.

- $f(n) = 3n^2 + 5n$ and $g(n) = n^3$. In this case

  $$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n^3} = \frac{3}{n} + \frac{5}{n^2} \longrightarrow 0 \text{ as } n \longrightarrow \infty$$

  This means that for large inputs the algorithm $G$ is doing a lot more work than $F$, making $F$ the more efficient algorithm.

In general, we can compare two complexity functions $f(n)$ and $g(n)$ by looking at what happens to $f(n)/g(n)$ as $n$ gets large. Although thinking in terms of a limit of this ratio is helpful in comparing the two algorithms, we cannot assume that such a limit will always exist. It turns out that a limit does not have to exist for us to gain useful information from this ratio: we can usefully compare the two complexity functions if we can find a positive constant $K$ such that

$$\frac{f(n)}{g(n)} \leq K \text{ for all } n \geq 1. \tag{16.1}$$

If this can be done, it means that the algorithm $F$ is no worse than $G$ for large problems, or in other words, $F$ is at least as good as $G$. In this case, we say that $f(n)$ is in $O(g(n))$, pronounced "$f$ is in Big O of $g$." The condition that defines $f(n)$ is in $O(g(n))$ is often written in the equivalent form shown in Equation (16.2).

$$f(n) \leq Kg(n) \text{ whenever } n \geq 1. \tag{16.2}$$

Showing that $f(n)$ is in $O(g(n))$ is usually straightforward: you look at the ratio $f(n)/g(n)$ and try to find a positive constant $K$ that satisfies Equation (16.1) for all $n \geq 1$. For example, to show that $3n^2 + 5n$ is in $O(n^2)$, look at the ratio

$$\frac{3n^2 + 5n}{n} = 3 + \frac{5}{n}$$

and notice that $5/n$ will be at most 5 for all $n \geq 1$. You can then use $K = 8$ in Equation (16.1).

To show that $f(n)$ is *not* in $O(g(n))$, you have to show that there is no way to find a positive $K$ that will satisfy Equation (16.1) for all $n \geq 1$. For example, the function $3n^2 + 5n$ is not in $O(n)$ because there is no constant $K$ that satisfies

$$\frac{3n^2 + 5n}{n} = 3n + 5 \leq K \text{ for all } n \geq 1.$$

Although defined for functions, the "Big O" notation and terminology is also used to characterize algorithms and computational problems. Thus, we say that an algorithm $F$ is in $O(g(n))$ for some function $g(n)$ if the complexity function $f(n)$ of $F$ is in Big O of $g(n)$. Accordingly, sequential search of an array is in $O(n)$ whereas binary search is in $O(\log n)$.

Similarly, a computational problem is said to be in $O(g(n))$ if there exists an algorithm for the problem whose worst case complexity function is in $O(g(n))$. Thus, the problem of sorting an array is in $O(n^2)$, whereas the problem of searching a sorted array is in $O(\log n)$.

If $g(n)$ is a function, $O(g(n))$ can be regarded as a family of functions that grow no faster than $g(n)$. These families are called complexity classes, and a few of them are important enough to merit specific names. We list them here in order of their rate of growth:

1. $O(1)$: A function $f(n)$ is in this class if there is a constant $K > 0$ such that $f(n) \leq K$ for all $n \geq 1$. An algorithm whose worst case complexity function is in this class is said to run in *constant time*.
2. $O(\log n)$: Algorithms in this class run in *logarithmic time*. Because $\log n$ grows much slower than $n$, a huge increase in the size of the problem results in a small increase in the running time of the algorithm. This complexity is characteristic of search problems that eliminate half of the search space with each operation. The *binary search* algorithm is in this class.
3. $O(n)$: Algorithms in this class run in *linear time*. Any increase in the size of the problem results in a proportionate increase in the running time of the algorithm. This complexity is characteristic of algorithms like *sequential search* that make a single pass, or a constant number of passes, over their input.
4. $O(n\log n)$: This class is called "$n \log n$" *time*. An increase in the size of the problem results in a slightly greater increase in the running time of the algorithm. The average case complexity of *Quicksort* lies in this class. Two well-known sorting algorithms, *Mergesort* and *Heapsort*, have worst case complexity functions that also lie in this class. We will discuss the *Heapsort* algorithm in Chapter 22.
5. $O(n^2)$: This class is called *quadratic time*. This performance is characteristic of algorithms that make multiple passes over the input data using two nested loops. An increase in the size of the problem causes a much greater increase in the running time of the algorithm. The worst case complexity functions of *bubble sort*, *selection sort*, *insertion sort*, and *Quicksort* all lie in this class.
6. The union of all classes $O(n^1)$, $O(n^2)$, $O(n^3)$, ... , forms a class called *polynomial time*.

## In the Spotlight:

### The Importance of Choosing an Efficient Algorithm

The algorithm selected for a particular task can have a significant impact on the running time of a program. As an example, let us consider the task of computing terms of the Fibonacci sequence. You first encountered this sequence in Chapter 15, and as you may recall, terms can be computed by this method:

```
public static long fib(long n)
{
   if (n <= 1)
       return 1;
   else
       return fib(n–2) + fib(n–1);
}
```

This implementation has the advantages of being short and easy to understand, and for that reason, it is often used to illustrate recursion. It is, however, very inefficient. We can get an idea of how inefficient it is by embedding it in a program that determines the time required to execute calls to the method. The following program allows a user to specify six consecutive values to be passed as arguments to the method and prints the time needed by the computer to execute each call.

**Code Listing 16-18** (RecursiveFibonacciTimer.java)

```
import java.util.Scanner;

/**
  This program times calls to the recursive Fibonacci method
  for 6 consecutive calls and displays the results.
*/

public class RecursiveFibonacciTimer
{
   public static void main(String[] args)
   {
      // Get the starting argument
      System.out.print("Enter a positive integer: ");
      Scanner sc = new Scanner(System.in);
      int number = sc.nextInt();

      // Variables used to determine time for a function call
      long currentTime = System.currentTimeMillis();
      long previousTime;
      long elapsedTime = 0;

      for (int k = 0; k <= 5; k++)
      {
         // Record time before calling the method
25          previousTime = currentTime;
26          System.out.print("The Fibonacci term at position ");
27          System.out.print((number + k) + " is " );
28
29          // Compute and print fib term for the next argument
30          System.out.println(fib(number + k));
31
32          // Record time after calling the method
33          currentTime = System.currentTimeMillis();
34
35          // Compute and print elapsed time in seconds
36          elapsedTime = (currentTime - previousTime) / 1000;
37          System.out.println("Computed in " + elapsedTime + " seconds.");
38       }
39    }
40
41    /**
42      Computes a term of the Fibonacci sequence
43      @param n
44      @return nth term of the sequence
45    */
46    public static long fib(long n)
47    {
48       if (n <= 1)
49          return 1;
50       else
51          return fib(n-2) + fib(n-1);
52    }
53 }
```

**Program Output with Example Input Shown in Bold**

```
Enter a positive integer: 45 [Enter]
The Fibonacci term at position 45 is 1836311903
Computed in 17 seconds.
The Fibonacci term at position 46 is 2971215073
Computed in 27 seconds.
The Fibonacci term at position 47 is 4807526976
Computed in 43 seconds.
The Fibonacci term at position 48 is 7778742049
Computed in 67 seconds.
The Fibonacci term at position 49 is 12586269025
Computed in 110 seconds.
The Fibonacci term at position 50 is 20365011074
Computed in 179 seconds.
```

In line 18, the program uses the `System.currentTimeMillis()` method to get the current time in milliseconds since midnight January 1, 1970. The value obtained is stored in the `currentTime` variable. At the beginning of each loop iteration, the current time is assigned to the `previousTime` variable in line 25. Then, after the call to the `fib` method, the current time is obtained again in line 33 and assigned to the `currentTime` variable. The time difference, in milliseconds, is calculated in line 36 by subtracting `previousTime` from `currentTime`. This difference is divided by 1000 to yield the time elapsed during the method call in seconds.

You can tell that the time required to compute a term of the sequence increases significantly for each successive term. For example, the computation of the term at position 45 requires a quarter of a minute, and five terms later, the 50th term requires nearly three minutes. And it gets worse from there. Clearly, this algorithm, although simple to write, is so inefficient as to be unusable for problems of moderately large sizes.

**Checkpoint**

16.9 What is a basic operation?

16.10 What is the worst case complexity function of an algorithm?

16.11 One algorithm needs $100n$ basic operations to process an input of size $n$, and another algorithm needs $25n$ basic operations to process the same input. Which of the two algorithms is more efficient?

16.12 What does it mean to say that $f(n)$ is not in $O(g(n))$?

16.13 What does it mean to say that $f(n)$ is in $O(g(n))$?

16.14 Show that $100n^3 + 50n^2 + 75$ is in $O(20n^3)$ by finding a positive $K$ that satisfies Equation (16.1) in this section.

16.15 Let $a[\ ]$ and $b[\ ]$ be two integer arrays of size $n$. Following the examples of this section, give a formal description of the problem of determining if every element of $a[\ ]$ is also an element of $b[\ ]$. The output of the algorithm should be one of the words "true" or "false."

16.16 Give an algorithm for solving the problem you defined in Checkpoint 16.15, and determine its worst case complexity.

16.17 Show that every function in $O(20)$ also is in $O(1)$.

16.18 Assuming $g(n) \geq 1$ for all $n \geq 1$, show that every function in $O(g(n) + 100)$ is also in $O(g(n))$.

## 16.4 Common Errors to Avoid

- **Using an inefficient sort or search algorithm on a large array.** Simple algorithms like the bubble sort and the sequential search are inefficient because they access array elements so many times. More efficient algorithms like Quicksort and the binary search are preferable, especially on large arrays.
- **Forgetting to sort the data in an array before using the binary search algorithm.** The binary search algorithm requires that its data already be sorted.
- **Using timing to determine the efficiency of an algorithm.** The results obtained from timing an application depend on many variables, including the programming language used and on the quality of the compiler used to generate machine code. A better approach is to count the number of basic steps an algorithm requires to process an input of a given size.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This search algorithm steps sequentially through an array, comparing each item with the search value.
   a. sequential search
   b. binary search
   c. natural order search
   d. selection search
2. This search algorithm repeatedly divides the portion of an array being searched in half.
   a. sequential search
   b. binary search
   c. natural order search
   d. selection search
3. This search algorithm will search half the array on average.
   a. sequential search
   b. binary search
   c. natural order search
   d. selection search
4. This search algorithm requires that the array’s contents be sorted.
   a. sequential search
   b. binary search
   c. natural order search
   d. selection search
5. If an array is sorted in this order, the values are stored from lowest to highest.
   a. asymptotic
   b. logarithmic
   c. ascending
   d. descending
6. If an array is sorted in this order, the values are stored from highest to lowest.
   a. asymptotic
   b. logarithmic
   c. ascending
   d. descending
7. This sorting algorithm makes several passes through an array and causes the larger values to gradually move toward the end of the array with each pass.
   a. bubble sort
   b. selection sort
   c. insertion sort
   d. Quicksort

8. This sorting algorithm recursively divides an array into sublists.
   a. bubble sort
   b. selection sort
   c. insertion sort
   d. Quicksort

9. In this sorting algorithm, the smallest value in the array is located and moved to position 0. Then the next smallest value is located and moved to position 1. This process continues until all of the elements have been placed in their proper order.
   a. bubble sort
   b. selection sort
   c. insertion sort
   d. Quicksort

10. This sorting algorithm begins by sorting the initial portion of the array consisting of two elements. Then, the third element is moved to its correct position, relative to the first two elements. At that point, the first three elements are in sorted order. This process continues with the fourth and subsequent elements until the entire array is sorted.
    a. bubble sort
    b. selection sort
    c. insertion sort
    d. Quicksort

11. **True or False:** If data is sorted in ascending order, it means it is ordered from lowest value to highest value.

12. **True or False:** If data is sorted in descending order, it means it is ordered from lowest value to highest value.

13. **True or False:** The *average* number of comparisons performed by the sequential search on an array of *n* elements is *n*/2.

14. **True or False:** The *maximum* number of comparisons performed by the sequential search on an array of *n* elements is *n*/2.

## Find the Error

15. Assume this code is using the IntBinarySearcher class presented in this chapter:

```
int[] numbers = { 8, 0, 9, 4, 3, 7, 2 };
// Search for 7.
int location = IntBinarySearcher.search(numbers, 7);
// Display the result.
if (location != −1)
   System.out.println("7 was found in the array.");
else
   System.out.println("7 was NOT found in the array.");
```

## Algorithm Workbench

1. Complete the following table calculating the average and maximum number of comparisons the sequential search will perform, and the maximum number of comparisons the binary search will perform.

| | Array Size: 50 elements | Array Size: 500 elements | Array Size: 10,000 elements | Array Size: 100,000 elements | Array Size: 10,000,000 elements |
|---|---|---|---|---|---|
| Sequential Search (Average Comparisons) | | | | | |
| Sequential Search (Maximum Comparisons) | | | | | |
| Binary Search (Maximum Comparisons) | | | | | |

## Short Answer

1. If a sequential search method is searching for a value that is stored in the last element of a 10,000-element array, how many elements will the search code have to examine to locate the value?
2. In an average case involving an array of *n* elements, how many array elements does a sequential search algorithm have to access to locate a specific value?
3. A binary search method is searching for a value that is stored in the middle element of an array. How many array elements have to be examined before finding the value?
4. What is the maximum number of comparisons that a binary search method will make when searching for a value in a 1,000-element array?
5. Why is the bubble sort inefficient for large arrays?
6. Under what conditions might the insertion sort be more efficient than the bubble sort?
7. List the steps that the selection sort algorithm would make in sorting the following values: 4, 1, 3, 2.
8. List the steps that the insertion sort algorithm would make in sorting the following values: 4, 1, 3, 2.

## Programming Challenges

### 1. Sorting Objects with the Insertion Sort Algorithm

The `insertionSort` method in the `IntInsertionSorter` class presented in this chapter sorts an array of `int` values. Create an `ObjectInsertionSorter` class that can sort `Comparable` objects. Demonstrate the class in a program that sorts an array of `String` objects.

### 2. Sorting Objects with the Quicksort Algorithm

The `IntQuickSorter` class presented in this chapter sorts an array of `int` values. Create an `ObjectQuickSorter` class that can sort `Comparable` objects. Demonstrate the class in a program that sorts an array of `String` objects.

VideoNote
Searching for Objects with the Binary Search Algorithm

### 3. Searching for Objects with the Binary Search Algorithm

The `IntBinarySearcher` class presented in this chapter searches an `int` array for a specific value. Create an `ObjectBinarySearcher` class that can search an array of `Comparable` objects. Demonstrate the class in a program that searches for a `String` in an array of `String` objects.

### 4. Charge Account Validation

Create a class with a method that accepts a charge account number as its argument. The method should determine whether the number is valid by comparing it to the following list of valid charge account numbers:

| | | | | | |
|---|---|---|---|---|---|
| 5658845 | 4520125 | 7895122 | 8777541 | 8451277 | 1302850 |
| 8080152 | 4562555 | 5552012 | 5050552 | 7825877 | 1250255 |
| 1005231 | 6545231 | 3852085 | 7576651 | 7881200 | 4581002 |

These numbers should be stored in an array. Use a sequential search to locate the number passed as an argument. If the number is in the array, the method should return `true`, indicating the number is valid. If the number is not in the array, the method should return `false`, indicating the number is invalid.

Write a program that tests the class by asking the user to enter a charge account number. The program should display a message indicating whether the number is valid or invalid.

### 5. Charge Account Validation Modification

Modify the program you wrote for Programming Challenge 4 (Charge Account Validation) so it performs a binary search to locate valid account numbers. Use your choice of sorting algorithm to sort the array before the binary search is performed.

### 6. Name Search

Download the Chapter 16 source code from www.pearson.com/gaddis, and, you will find a text file by the name of *names.txt*. This file contains a list of the 65 most popular female first names in the United States according to the 1990 census. Write a program that reads these names into a `String` array. Use the Quicksort algorithm to sort the names in ascending order. Then allow the user to search for a name in the array. Use the binary search algorithm to perform the search.

## 7. Search Benchmarks

Write an application that has an array of at least 20 integers. It should call a method that uses the sequential search algorithm to locate one of the values. The method should keep a count of the number of comparisons it makes until it finds the value. Then the program should call another method that uses the binary search algorithm to locate the same value. It should also keep count of the number of comparisons it makes. Display these values on the screen.

## 8. Sorting Benchmarks

Modify the methods presented in this chapter that perform the bubble sort, selection sort, insertion sort, and Quicksort algorithms on an `int` array, such that each method keeps a count of the number of swaps it makes.

Then, write an application that uses four identical arrays of at least 20 integers. It should call each method on a different array, and display the number of exchanges made by each algorithm.

## 9. Efficient Computation of Fibonacci Numbers

Modify the application in Code Listing 16-18 by replacing the recursive method for computing Fibonacci terms with an iterative version and try to make it as efficient as possible. Modify the application so it outputs the elapsed time in both seconds and milliseconds. Finally, run the application and generate output showing the time required to compute the terms of the sequence in positions 45, 46, 47, 48, 49, and 50. Comment on the results.

# CHAPTER 17 Generics

## TOPICS

17.1 Introduction to Generics
17.2 Writing a Generic Class
17.3 Passing Objects of a Generic Class to a Method
17.4 Writing Generic Methods
17.5 Constraining a Type Parameter in a Generic Class
17.6 Inheritance and Generic Classes
17.7 Defining Multiple Type Parameters
17.8 Generics and Interfaces
17.9 Erasure
17.10 Restrictions on the Use of Generic Types
17.11 Common Errors to Avoid

## 17.1 Introduction to Generics

**CONCEPT: A generic class or method is one that permits you to specify the allowable types of objects that the class or method may work with. If you attempt to use the class or method with an object that is not of an allowable type, an error occurs at compile time.**

In Chapter 7, you were introduced to the `ArrayList` class, which is in the `java.util` package. Once you create an `ArrayList` object, you can use it as a container to hold other objects. An `ArrayList` object automatically expands in size to accommodate the number of objects that are stored in it, which is an advantage over traditional arrays. The `ArrayList` class can be used in such a way that a mixture of objects, of any type, can be simultaneously stored in it. For example, look at Code Listing 17-1. This program stores a `String` object, a `Double` object, and a `Rectangle` object in the same `ArrayList`.

**Code Listing 17-1** (`UnsafeArrayList.java`)

```
import java.util.ArrayList;

/**
   This program demonstrates that an ArrayList can accept
   a mixture of object types as its elements.
*/
```

```
 7
 8  public class UnsafeArrayList
 9  {
10     public static void main(String[] args)
11     {
12        // Create an ArrayList object.
13        ArrayList list = new ArrayList();
14
15        // Store a variety of objects in the list.
16        list.add("Java is fun!");         // Add a String
17        list.add(new Double(2.5));        // Add a Double
18        list.add(new Rectangle(10, 12));  // Add a Rectangle
19
20        // Retrieve a reference to each object in the list.
21        // Note that the reference returned from the get
22        // method must be cast to the correct type.
23        String s = (String)list.get(0);
24        Double d = (Double)list.get(1);
25        Rectangle r = (Rectangle)list.get(2);
26     }
27  }
```

In lines 16, 17, and 18 the `ArrayList` class's `add` method is called to add a `String`, a `Double`, and a `Rectangle` to the `ArrayList`. In lines 23, 24, and 25 the `get` method is called to get a reference to each object stored in the `ArrayList`. The statement in line 23 gets a reference to element 0 (the `String`), line 24 gets a reference to element 1 (the `Double`), and line 25 gets a reference to element 2 (the `Rectangle`).

The `get` method returns an `Object` reference, so it is necessary to cast the method's return value to the desired type. Line 23 casts the method's return type to a `String` reference, line 24 casts it to a `Double` reference, and line 25 casts it to a `Rectangle` reference. If we do not cast the `get` method's return value, an error will occur at compile time. This is because the compiler does not know the exact type of the object being returned; only that it is an `Object`.

Even though the cast operator prevents an error from occurring at compile time, it does not prevent an error at runtime. We can still have problems if we inadvertently cast the `get` method's return value to the wrong type. For example, look at the following code:

```
12  // Create an ArrayList object.
13  ArrayList list = new ArrayList();
14
15  // Store a variety of objects in the list.
16  list.add("Java is fun!");        // Store a String at element 0
17  list.add(new Double(2.5));       // Store a Double at element 1
18  list.add(new Rectangle(10, 12)); // Store a Rectangle at element 2
19
20  // Retrieve a reference to each object in the list.
21  Double d = (Double)list.get(0);  // Error! Element 0 is a String
```

```
22    String s = (String)list.get(1);
23    Rectangle r = (Rectangle)list.get(2);
```

This code starts out just like the `main` method in Code Listing 17-1. An `ArrayList` is created, and then a `String`, a `Double`, and a `Rectangle` are stored in it. An error occurs in line 21, however. This statement gets a reference to the object at element 0. The problem is that it casts the return value of the `get` method to a `Double`, when the object that is actually returned is a `String`. The error that results from this statement does not happen at compile time, however. It happens at runtime. When the code runs, a `ClassCastException` will be thrown.

Fortunately, the Java language provides a feature known as generic classes and methods, commonly referred to as generics. *Generics* allow the compiler to perform type-checking and report type incompatibilities. Rather than waiting for your finished application to throw an exception, it is far better to discover at compile time the places in your code where types have been mismatched.

A generic class or method is one that permits you to specify the allowable types of objects that the class or method may work with. If you attempt to use the class or method with an object that is not of an allowable type, an error occurs at compile time. Recall from Chapter 7 that the `ArrayList` class can also be used as a generic class. It allows you to specify the type of object that you want to store in an `ArrayList`, and the Java compiler will make sure that only objects of the specified type are stored in it. For example, we can create an `ArrayList` that will hold only `String` objects with a statement such as the following:

```
ArrayList<String> myList = new ArrayList<String>();
```

This statement creates an `ArrayList` object referenced by the variable `myList`. Because we have written the notation `<String>` immediately after the word `ArrayList`, we will only be allowed to store `String` objects in the `ArrayList`. If we try to store any other type of object in the `ArrayList`, an error will occur at compile time.

**TIP:** The notation `ArrayList<String>` is pronounced "`ArrayList` of `String`."

Similarly, if we wish to create an `ArrayList` object that can hold only `Student` objects, we would use a statement such as the following:

```
ArrayList<Student> myStudents = new ArrayList<Student>();
```

If we write a statement that stores an object of any type other than `Student` in `myStudents`, the compiler will generate an error. In these two examples, we can provide the notation `<String>` or `<Student>` because the `ArrayList` class can be used as a generic class.

Because the compiler knows the type of every element, we do not have to use a cast operator when retrieving an element from a generic `ArrayList`. For example, the following code creates an `ArrayList` that can hold only `String` objects, stores a string in it, and then retrieves the string. Notice that the last statement does not use a cast operator.

```
ArrayList<String> myList = new ArrayList<String>();
myList.add("Java is fun!");
String str = myList.get(0);
```

## Using the Diamond Operator for Type Inference

Java allows you to use the diamond operator ( <> ) to simplify the instantiation of a generic class. Previously, you saw the following statement that creates an `ArrayList` object:

```
ArrayList<String> list = new ArrayList<String>();
```

Notice that the data type `String` is written between the angled brackets in two locations: first in the part that declares the reference variable (on the left side of the = operator), and then again in the part that calls the `ArrayList` constructor (on the right side of the = operator). You can use the diamond operator to simplify this statement, as follows:

```
ArrayList<String> list = new ArrayList<>();
```

By using the diamond operator, you are not required to write the data type in the part of the statement that calls the `ArrayList` constructor. The diamond operator causes the compiler to infer the required data type from the reference variable declaration. Here is another example:

```
ArrayList<Student> list = new ArrayList<>();
```

Assuming that we have a class named `Student`, this statement creates an `ArrayList` that can hold `Student` objects.

**Checkpoint**

17.1 Why is it risky to store objects of different types in an `ArrayList`?

17.2 When `ArrayList` is used as a non-generic class, why must you use the cast operator with the `get` method?

17.3 Suppose we use the following statement to instantiate the `ArrayList` class. What type of objects will we be allowed to store in the `ArrayList`?

```
ArrayList<Rectangle> myList = new ArrayList<>();
```

17.4 Assume we have used the statement shown in Checkpoint 17.3 to instantiate the `ArrayList` class. If we attempt to store an object that is not of the allowable type in the `ArrayList`, will we see an error message at compile time or at runtime?

## 17.2 Writing a Generic Class

**CONCEPT: In Java you can write your own generic classes that accept data type arguments.**

Not only does Java provide several generic classes for you to use, such as `ArrayList`, but it also allows you to write your own generic classes. Code Listing 17-2 shows the `Point` class, which is a simple example of a generic class. (This version of the class is stored in the source code folder *Chapter 17\Point Class Version 1*.) An instance of the class holds two values: an *X* coordinate and a *Y* coordinate.

**Code Listing 17-2** (`Point.java`)

```
/**
   The Point class holds X and Y coordinates. The data type
   of the coordinates is generic.
*/

public class Point<T>
{
   private T xCoordinate;  // The X coordinate
   private T yCoordinate;  // The Y coordinate

   /**
      Constructor
      @param x The X coordinate.
      @param y The Y coordinate.
   */

   public Point(T x, T y)
   {
      xCoordinate = x;
      yCoordinate = y;
   }

   /**
      The setX method sets the X coordinate.
      @param x The value for the X coordinate.
   */

   public void setX(T x)
   {
      xCoordinate = x;
   }

   /**
      The setY method sets the Y coordinate.
      @param y The value for the Y coordinate.
   */

   public void setY(T y)
   {
      yCoordinate = y;
   }

   /**
      The getX method returns the X coordinate.
      @return The value of the X coordinate.
   */
47
48     public T getX()
49     {
50        return xCoordinate;
51     }
52
53     /**
54        The getY method returns the Y coordinate.
55        @return The value of the Y coordinate.
56     */
57
58     public T getY()
59     {
60        return yCoordinate;
61     }
62  }
```

Let's take a closer look at the class header in line 6:

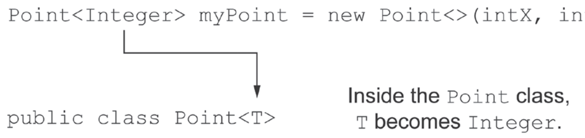
```
public class Point<T>
```

After the name of the class, `T` appears inside angled brackets. In this class declaration, `T` is a type parameter. A *type parameter* is an identifier that represents an actual type. When you instantiate the class, you use an actual type as an argument for the type parameter. You do this by writing the type name inside angled brackets, immediately following the name of the class. For example, consider the following code:

```
Integer intX = new Integer(10);
Integer intY = new Integer(20);
Point<Integer> myPoint = new Point<>(intX, intY);
```

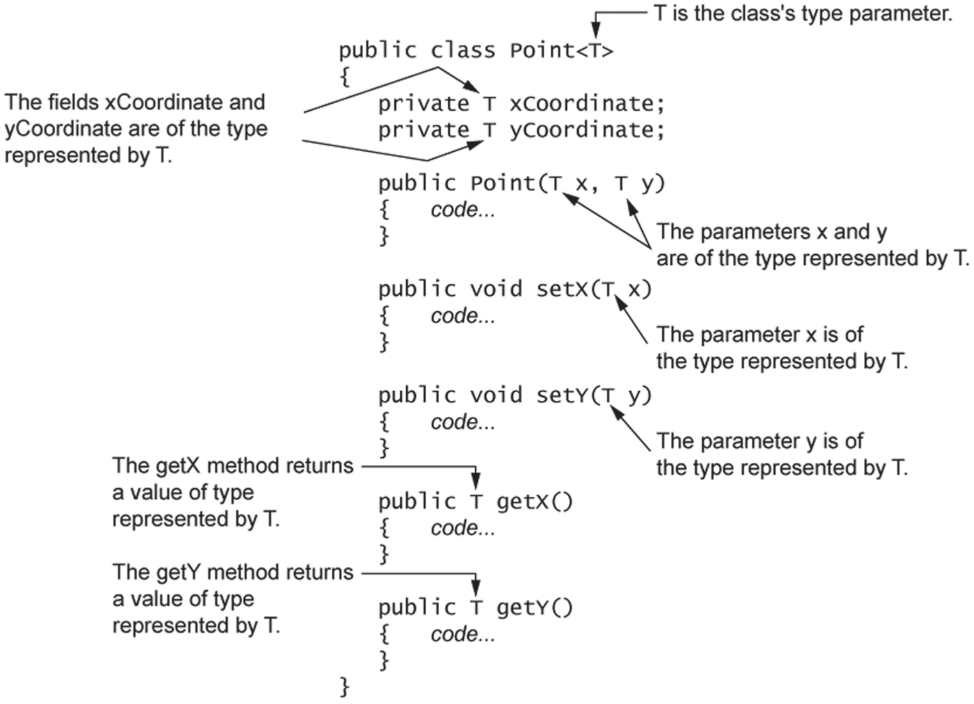
Recall that `Integer` is a wrapper class. In this statement we are passing `Integer` as a type argument to the `T` parameter. This is illustrated in Figure 17-1.

**Figure 17-1** Passing a type argument to a type parameter



For this instance of the `Point` class, the type parameter `T` will become `Integer`. There are several places in the `Point` class where `T` is used, as shown in Figure 17-2. In each place, `T` will become the type that was passed to it as an argument.

**Figure 17-2** The type parameter T in the `Point` class


T is the class's type parameter.
public class Point<T>
{
The fields xCoordinate and yCoordinate are of the type represented by T.
private T xCoordinate;
private T yCoordinate;
public Point(T x, T y)
{ code...
}
The parameters x and y are of the type represented by T.
public void setX(T x)
{ code...
}
The parameter x is of the type represented by T.
public void setY(T y)
{ code...
}
The parameter y is of the type represented by T.
The getX method returns a value of type represented by T.
public T getX()
{ code...
}
The getY method returns a value of type represented by T.
public T getY()
{ code...
}
}

For example, look at the following declarations of the `xCoordinate` and `yCoordinate` fields in lines 8 and 9:

```
private T xCoordinate;         // The X coordinate
private T yCoordinate;         // The Y coordinate
```

If we pass `Integer` as a type argument to `T`, these declarations will become as follows:

```
private Integer xCoordinate;   // The X coordinate
private Integer yCoordinate;   // The Y coordinate
```

The `T` type parameter is also used in the following method headers:

```
public Point(T x, T y)
public void setX(T x)
public void setY(T y)
public T getX()
public T getY()
```

If we pass `Integer` as a type parameter to `T`, these method headers will become as follows:

```
public Point(Integer x, Integer y)
public void setX(Integer x)
public void setY(Integer y)
public Integer getX()
public Integer getY()
```

The program in Code Listing 17-3 tests the `Point` class by creating two instances of it. In the first instance, the `Double` type is passed to the type parameter `T`, and in the second instance the `Integer` type is passed to `T`. (This program is also stored in the source code folder *Chapter 17\Point Class Version 1*.)

**Code Listing 17-3** (`TestPoint.java`)

```
/**
   This program demonstrates the Point class.
*/

public class TestPoint
{
   public static void main(String[] args)
   {
      // Create two Double objects to use as coordinates.
      Double dblX = new Double(1.5);
      Double dblY = new Double(2.5);

      // Create a Point object that can hold Doubles.
      Point<Double> dPoint = new Point<>(dblX, dblY);

      // Create two Integer objects to use as coordinates.
      Integer intX = new Integer(10);
      Integer intY = new Integer(20);

      // Create a Point object that can hold Integers.
      Point<Integer> iPoint = new Point<>(intX, intY);

      // Display the Double values stored in dPoint.
      System.out.println("Here are the values in dPoint.");
      System.out.println("X Coordinate: " + dPoint.getX());
      System.out.println("Y Coordinate: " + dPoint.getY());
      System.out.println();

      // Display the Integer values stored in iPoint.
      System.out.println("Here are the values in iPoint.");
      System.out.println("X Coordinate: " + iPoint.getX());
      System.out.println("Y Coordinate: " + iPoint.getY());
   }
}
```

**Program Output**

```
Here are the values in dPoint.
X Coordinate: 1.5
Y Coordinate: 2.5
Here are the values in iPoint.
X Coordinate: 10
Y Coordinate:  20
```

When the following statement is executed in line 14, the `Double` type is passed as an argument to the type parameter `T`:

```
Point<Double> dPoint = new Point<>(dblX, dblY);
```

For this particular instance of the `Point` class, the `T` type parameter is replaced with `Double`. The method header for this particular object's constructor becomes the following:

```
public Point(Double x, Double y)
```

Notice that in line 14 references to two `Double` objects, `dblX` and `dblY`, are passed as arguments to the constructor. If we had passed anything other than `Double` references, the compiler would have generated an error.

Another instance of the class is created with the following statement in line 21:

```
Point<Integer> iPoint = new Point<>(intX, intY);
```

For this particular instance of the `Point` class, the `T` type parameter is replaced with `Integer`. The method header for this particular object's constructor becomes the following:

```
public Point(Integer x, Integer y)
```

In this statement we have passed two `Integer` references, `intX` and `intY`, as arguments to the constructor. If we had passed anything other than `Integer` references, an error would have occurred at compile time.

## Only Reference Types Can Be Passed to Type Parameters

When you create an instance of a generic class, you can only pass a reference type to the class's type parameter. This is the reason that in Code Listing 17-3 we passed the `Double` and `Integer` types to the `Point` class instead of the primitive `double` and `int` types. The only way to use a generic type to hold a primitive value is to wrap the primitive value up in a wrapper class, as we demonstrated in Code Listing 17-3.

### Autoboxing and Unboxing

Although we can only pass reference types to a generic class's type parameter, we can use Java's autoboxing and unboxing feature to simplify our code if we need to store primitive values in a generic type. We briefly introduced autoboxing and unboxing in Chapter 9. Recall that *autoboxing* is Java's process of automatically "boxing up" a primitive value inside an object of an appropriate wrapper class. For example, the statement in line 14 of Code Listing 17-3 could have been written as follows:

```
Point<Double> dPoint = new Point<>(1.5, 2.5);
```

Because the `Point` constructor accepts two `Double` arguments, the literal values 1.5 and 2.5 are autoboxed into two objects of the `Double` class. References to those objects are then passed as arguments to the constructor.

*Unboxing* is the opposite of boxing. It is the process of automatically converting a wrapper class object to a primitive type, when needed. For example, look at the following code:

```
// Use autoboxing to pass doubles to the constructor.
Point<Double> dPoint = new Point<>(1.5, 2.5);
```

```
// Use unboxing to retrieve the X and Y coordinates
// and assign them to double variables.
double x = dPoint.getX();
double y = dPoint.getY();
```

In the last two statements, the `getX` and `getY` methods are called, which will return references to `Double` objects. Because the methods' return values are being assigned to `double` variables, however, Java will unbox the `Double` objects and store their values in the `double` variables. After this code executes the variables `x` and `y` will hold the values 1.5 and 2.5.

## Instantiating a Generic Class without Specifying a Type Argument

Although it is not recommended, it is possible to create an instance of a generic class without providing a type argument. When you do this, you are using the class as a *raw type*. For example, look at the program in Code Listing 17-4. (This program is stored in the source code folder *Chapter 17\Point Class Version 1*.)

**Code Listing 17-4** (`TestPoint2.java`)

```
/**
   This demonstrates the Point class without a type argument.
*/

public class TestPoint2
{
   public static void main(String[] args)
   {
      // Create an Integer and a Double
      // to use as coordinates.
      Integer x = new Integer(1);
      Double y = new Double(7.5);

      // Create a Point object with an
      // Integer x coordinate and a Double
      // y coordinate.
      Point myPoint = new Point(x, y);

      // Display the object's X and Y
      // coordinate values.
      System.out.println("X Coordinate: " +
                         myPoint.getX());
      System.out.println("Y Coordinate: " +
                         myPoint.getY());
   }
}
```

**Program Output**

```
X Coordinate: 1
Y Coordinate: 7.5
```

An instance of the `Point` class is created with the following statement in line 17:

```
Point myPoint = new Point(x, y);
```

Notice that there is no type argument provided in angled brackets. When a type argument is not provided for an instance of a generic class, Java will provide `Object` as the type argument by default. This means that for the `myPoint` object, the declarations for the `xCoordinate` and `yCoordinate` fields will become the following:

```
private Object xCoordinate;    // The X coordinate
private Object yCoordinate;    // The Y coordinate
```

Likewise, each occurrence of `T` in the class methods will become `Object`.

**NOTE:** If you compile the *TestPoint2.java* program shown in Code Listing 17-4 you will see a warning such as the following:

```
TestPoint2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The Java compiler issues this warning when we compile code that instantiates a generic class without specifying a type argument.

It is not recommended that you instantiate a generic class without providing a type argument, because you give up the benefits of type-safety that you would have otherwise. For example, in line 17 of Code Listing 17-4, an `Integer` object is passed to the `Point` class constructor as the *X* coordinate, and a `Double` object is passed as the *Y* coordinate. This is legal because the constructor's parameter variables are of the `Object` type, and we can polymorphically pass references of any type to them.

It also means that the class's `xCoordinate` and `yCoordinate` fields can reference objects of any type. This opens up the door for runtime errors if we attempt to perform an unsupported operation on one of the class's fields.

Another reason that we should avoid instantiating a generic class without providing a type argument is that we have to use a cast operator to assign an `Object` reference to a variable of any other type. The following code illustrates this:

```
// Create an Integer and a Double.
Integer x = new Integer(1);
Double y = new Double(7.5);

// Create a Point object with no type argument.
Point myPoint = new Point(x, y);

// Retrieve the X and Y coordinates.
Integer xx = (Integer) myPoint.getX();
Double yy = (Double) myPoint.getY();
```

The last two statements shown here use the `myPoint` object to call the `getX` and `getY` methods. Because these methods return an `Object` reference, we must cast the reference to the proper type in order to assign it to a variable. This is not necessary when we instantiate the `Point` class with a type argument.

## Commonly Used Type Parameter Names

The generic `Point` class that we examined defines a type parameter named `T`. There's nothing magic about the name `T`. When you are defining a generic type parameter, you can use any legal identifier you wish. By convention, however, type parameters are usually single letters written in uppercase. This helps distinguish the name of a type parameter from the name of a class. Also, there are some names that are commonly used, depending on the purpose of the type parameter. Table 17-1 lists the commonly used parameter names and briefly describes their meaning.

**Table 17-1** Common Type Parameter Names

| Name | Usual Meaning |
|---|---|
| `T` | `T` is commonly used for a general type. |
| `S` | `S` is commonly used for a general type when `T` has already been used. |
| `E` | `E` is commonly used to represent the type of an element, such as an element in an `ArrayList` or other collection. |
| `K` | `K` is commonly used to represent the type of a key in a class that keeps key/value pairs. |
| `V` | `V` is commonly used to represent the type of a value in a class that keeps key/value pairs. |

As we progress through this chapter you will see several of these type parameter names being used.

**Checkpoint**

17.5 What is a type parameter?

17.6 When you create an instance of a generic class, can you pass reference types, primitive types, or either to the class's type parameter?

17.7 Is it possible to create an instance of a generic class without providing a type argument? If so, is this recommended?

17.8 Is it required that type parameters be single letters written in uppercase?

## 17.3 Passing Objects of a Generic Class to a Method

**CONCEPT: When you create instances of a generic class, you must be able to pass those instances as arguments to methods. Method parameter variables can be declared to be of a generic type.**

Suppose you want to write a method that accepts an instance of a generic class as an argument. For example, the following `printPoint` method accepts an instance of `Point<Integer>` as an argument and displays the *X* and *Y* coordinate values in that object.

```
public static void printPoint(Point<Integer> point)
{
   System.out.println("X Coordinate: " + point.getX());
   System.out.println("Y Coordinate: " + point.getY());
}
```

In this method, the parameter `point` is a reference to a `Point<Integer>` object. We can call the method and pass a reference to any `Point<Integer>` object, as shown here:

```
Point<Integer> iPoint = new Point<>(7, 12);
printPoint(iPoint);
```

A problem arises, however, if we want to pass an instance of `Point<Double>` to the method. The method's parameter is a reference to `Point<Integer>`, so only `Point<Integer>` objects can be passed to it.

If the `printPoint` method is intended to be a general purpose method that can display the *X* and *Y* coordinates of any `Point` object, this limitation will be too restrictive. It seems reasonable that any numeric wrapper class could be used as a valid type argument when creating an instance of the `Point` class. For example, we should expect that instances of `Point<Integer>`, `Point<Double>`, `Point<Long>`, `Point<Float>`, and `Point<Short>` are all valid. In that case, we would want the `printPoint` method to be able to accept instances of any of these types as an argument.

At this point you know something about inheritance and polymorphism, and you might be thinking along these lines: "I just checked the Java API documentation and found that all of the numeric wrapper classes extend the `Number` class, which is in the `java.lang` package. Why not use `Number` as the type argument in the method's parameter? That way we can use polymorphism to accept any `Point` object whose type argument extends the `Number` class."

In other words, why not write the `printPoint` method like this?

```
// Will this do what we want?
public static void printPoint(Point<Number> point)
{
   System.out.println("X Coordinate: " + point.getX());
   System.out.println("Y Coordinate: " + point.getY());
}
```

Unfortunately, this will not give us the results we want. The Java compiler will allow us to pass *only* `Point<Number>` objects to this method. If we try to pass a `Point<Integer>` or `Point<Double>` object, we will get an error at compile time. One way to solve our problem is to use the `?` type wildcard with the parameter, as shown here:

```
public static void printPoint(Point<?> point)
{
   System.out.println("X Coordinate: " + point.getX());
   System.out.println("Y Coordinate: " + point.getY());
}
```

Notice that the parameter's type is `Point<?>`. The `?` character that appears inside the angled brackets is a *type wildcard*, and it indicates that any type argument can be used in its place.

Now we can pass any `Point` object as an argument to the method, regardless of its type argument. The following statements will compile and execute without errors:

```
Point<Integer> iPoint = new Point<>(1, 2);
Point<Double> dPoint = new Point<>(1.5, 2.5);
printPoint(iPoint);
printPoint(dPoint);
```

## Constraining a Type Parameter

Recall that the header for our original version of the `printPoint` method header looked like the following:

```
public static void printPoint(Point<Integer> point)
```

Because the parameter's type, `Point<Integer>`, was too restrictive, we changed the method header to the following:

```
public static void printPoint(Point<?> point)
```

The method now allows us to pass any `Point` object as an argument. However, this leads to a different problem: The parameter's new type, `Point<?>`, might not be restrictive enough. Suppose we only want to accept `Point` objects whose type argument is a numeric wrapper class. In other words, we only want to accept `Point` objects whose type argument is a subclass of `Number`. This can be accomplished by modifying the method once more, as shown here:

```
public static void printPoint(Point<? extends Number> point)
{
   System.out.println("X Coordinate: " + point.getX());
   System.out.println("Y Coordinate: " + point.getY());
}
```

In this version of the method, the parameter's type is `Point<? extends Number>`. This means that the `Point` object's type argument may be `Number`, or any type that extends `Number`. The following statements will compile and execute without errors:

```
Point<Integer> iPoint = new Point<>(1, 2);
Point<Double> dPoint = new Point<>(1.5, 2.5);
printPoint(iPoint);
printPoint(dPoint);
```

However, the following code will not compile because the `String` class is not a subclass of `Number`:

```
Point<String> sPoint = new Point<>("1", "2");
printPoint(sPoint);    // Error!
```

Code Listing 17-5 shows the *TestPoint3.java* program, which demonstrates this version of the `printPoint` method. (This program is stored in the source code folder *Chapter 17\Point Class Version 1*.)

**Code Listing 17-5** (`TestPoint3.java`)

```java
/**
   This program demonstrates the printPoint method.
*/

public class TestPoint3
{
   public static void main(String[] args)
   {
      // Create various Point objects.
      Point<Integer> iPoint = new Point<>(1, 2);
      Point<Double> dPoint = new Point<>(1.5, 2.5);
      Point<Long> lPoint = new Point<>(10L, 20L);
      Point<Float> fPoint = new Point<>(7.9f, 9.9f);

      // Display each object's coordinates.
      System.out.println("iPoint:");
      printPoint(iPoint);

      System.out.println("\ndPoint:");
      printPoint(dPoint);

      System.out.println("\nlPoint:");
      printPoint(lPoint);

      System.out.println("\nfPoint:");
      printPoint(fPoint);
   }

   /**
      The printPoint method prints a Point object's
      X and Y coordinates.
      @param point A Point object.
   */

   public static void printPoint(Point<? extends Number> point)
   {
      System.out.println("X Coordinate: " + point.getX());
      System.out.println("Y Coordinate: " + point.getY());
   }
}
```

**Program Output**

```
iPoint:
Coordinate: 1
Coordinate: 2
```

```
dPoint:
Coordinate: 1.5
Coordinate: 2.5

lPoint:
Coordinate: 10
Coordinate: 20

fPoint:
Coordinate: 7.9
Coordinate: 9.9
```

**NOTE:** In line 12 the values being passed as arguments to the `Point` class constructor are `10L` and `20L`. The “L” suffix tells the compiler to treat these literal values as long integers. Without the “L” suffix, these literals would be treated as `int` values, and an error would occur. Similarly, in line 13 the values `7.9f` and `9.9f` are being passed as arguments to the constructor. The “f” suffix tells the compiler to treat these literals as `float` values. Without the “f” suffix, these literals would be treated as `double` values and an error would occur.

## Defining a Type Parameter in a Method Header

In the previous example you saw how the notation `<? extends Number>` means “any type that is `Number` or a subclass of `Number`.” As an alternative to using the wildcard in this notation, you can actually define a type parameter in a method header. For example, the `printPoint` method could have been written as follows:

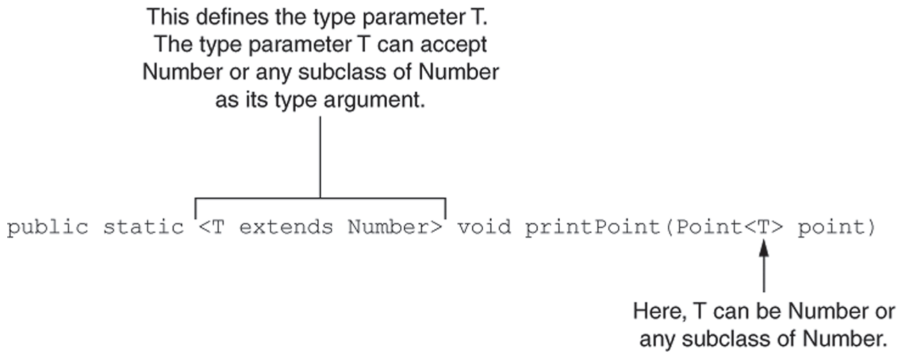
```
public static <T extends Number> void printPoint(Point<T> point)
{
   System.out.println("X Coordinate: " + point.getX());
   System.out.println("Y Coordinate: " + point.getY());
}
```

Notice that between the words `static` and `void`, the notation `<T extends Number>` appears. This defines a type parameter named `T`, and specifies that `T` can accept any type that is `Number` or a subclass of `Number`. Then notice that the type of the method’s parameter is `Point<T>`. This means that the method can accept any `Point` object whose type argument is `Number` or a subclass of `Number`. This is illustrated in Figure 17-3.

Using this alternative syntax can simplify methods that accept multiple arguments of generic types. For example, look at the following method header:

```
public static void doSomething(Point<? extends Number> arg1,
                               Point<? extends Number> arg2,
                               Point<? extends Number> arg3)
```

**Figure 17-3** Type parameter defined in a method header


This defines the type parameter T.
The type parameter T can accept
Number or any subclass of Number
as its type argument.
public static <T extends Number> void printPoint(Point<T> point)
Here, T can be Number or
any subclass of Number.

This method accepts three arguments, each of which is a `Point<? extends Number>` object. We had to write `Point<? extends Number>` as the type for each parameter variable. We can simplify the notation by defining a type parameter, as shown here:

```
public static <T extends Number> void doSomething(Point<T> arg1,
                                                  Point<T> arg2,
                                                  Point<T> arg3)
```

## The `extends` Key Word Constrains a Type to an Upper Bound

When you use the `extends` key word with a generic type parameter, as we have shown in this section, it is said that you are constraining the type parameter to an *upper bound*. For example, in the notation `<T extends Number>` we are constraining `T` to the upper bound `Number`. This means that `T` can be any type that is below `Number` in the class hierarchy (or `Number` itself), but it cannot be a type that is above `Number` in the class hierarchy.

## The `super` Key Word Constrains a Type to a Lower Bound

In addition to the `extends` key word, you can use the `super` key word to restrict a type parameter. Here is an example of how it might be used in a method header:

```
public static void doSomething(Point<? super Integer> arg)
```

In this method, the `arg` parameter's type is `Point<? super Integer>`. This means that the `Point` object's type argument may be `Integer`, or any superclass of `Integer`. Because the type can be any class that appears above `Integer` in the class hierarchy, it is said that we are constraining the type to a *lower bound*. The `Integer` class has only two superclasses: `Number` and `Object`. So, this method will only accept as arguments objects that are `Point<Integer>`, `Point<Number>`, or `Point<Object>`.

**NOTE:** You can use either `extends` to constrain a type to an upper bound, or `super` to constrain a type to a lower bound, but not both.

**Checkpoint**

17.9 In generic type notation, what does the `?` character (the type wildcard) indicate?

17.10 In generic type notation, what does `<? extends Number>` mean?

17.11 In generic type notation, what does `<? super Integer>` mean?

17.12 In generic type notation, does the `extends` key word establish an upper boundary or a lower boundary?

17.13 In generic type notation, does the `super` key word establish an upper boundary or a lower boundary?

17.14 In generic type notation, can you use both `extends` and `super` to constrain a type to a set of boundaries?

## 17.4 Writing Generic Methods

**CONCEPT: In addition to classes, methods can also be generic.**

Previously, you saw how a method can accept an object of a generic class as an argument. Methods themselves can also be generic. This means that they can have their own type parameters, and can use those type parameters to represent the types of arguments, the types of local variables, and their return type. Code Listing 17-6 shows a class with a generic method.

**Code Listing 17-6** `(GenericMethodDemo.java)`

```
/**
   This program demonstrates a simple generic method.
*/

public class GenericMethodDemo
{
   public static void main(String[] args)
   {
      String[] names = { "Alfonso", "Beatrice", "Celine" };
      displayArray(names);
   }

   /**
      The displayArray method displays each element
      in an array.
      @param array The array to display.
   */

   public static <E> void displayArray(E[] array)
   {
      for (E element : array)
22             System.out.println(element);
23       }
24    }
```

**Program Output**

```
Alfonso
Beatrice
Celine
```

Line 19 shows the header for the `displayArray` method. Notice that the method defines a type parameter named `E`. The method has a parameter variable, `array`, which is a reference to an array of `E` objects. The `E` type parameter is also used in line 21, in the enhanced `for` loop. Notice in line 10 that the `displayArray` method is called like a regular method. Even though it is a generic method, no type argument is passed to it. When you call a generic method, the compiler determines which type to use from the context in which you are using the method. For example, in line 10 we are passing `names` as an argument to the method. Because `names` is a `String` array, the compiler uses the `String` type for `E` in the method.

You can constrain a type parameter in a generic method. For example, suppose the `displayArray` method had been written as follows:

```
public static <E extends Number> void displayArray(E[] array)
{
   for (E element : array)
      System.out.println(element);
}
```

In this version of the method, we have constrained the type parameter `E` to any type that is `Number` or a subclass of `Number`. When calling this version of the method, we can only pass an array of `Number` objects, or of objects of a subclass of `Number`.

**Checkpoint**

17.15 When an instance of a generic class is created, a type argument is explicitly passed to the class's type parameter. When a generic method is called, how is a type argument provided for the method's type parameter?

## 17.5 Constraining a Type Parameter in a Generic Class

**CONCEPT: You can constrain the type parameter in a generic class so that only certain type arguments can be used to create an instance of the class.**

Previously you saw how we used generic notation in the `printPoint` method header so the method can accept an instance of the generic `Point` class as an argument. You also saw how we used generic notation to constrain the method so that it will only accept `Point` objects whose type argument is `Number` or a subclass of `Number`.

Suppose we wish to constrain the `Point` class itself so that only certain type arguments can be used to create an instance of the class. For example, suppose we want to allow instances of the `Point` class to be created using only the numeric wrapper classes as type arguments. That would prevent a programmer from using classes such as `String` as the type argument when instantiating the `Point` class. We can impose such a restriction by using the `extends` key word to constrain the class's type parameter.

Code Listing 17-7 shows a modified version of the `Point` class, which uses the `extends` key word in its generic type notation. (This version of the class is stored in the source code folder *Chapter 17\Point Class Version 2*.)

**Code Listing 17-7** (`Point.java`)

```
/**
   The Point class holds X and Y coordinates. The data type
   of the coordinates is generic, but restricted to
   Number or a subclass of Number.
*/

public class Point<T extends Number>
{
   private T xCoordinate;  // The X coordinate
   private T yCoordinate;  // The Y coordinate

   /**
      Constructor
      @param x The X coordinate.
      @param y The Y coordinate.
   */

   public Point(T x, T y)
   {
      xCoordinate = x;
      yCoordinate = y;
   }

   /**
      The setX method sets the X coordinate.
      @param x The value for the X coordinate.
   */

   public void setX(T x)
   {
      xCoordinate = x;
   }

   /**
      The setY method sets the Y coordinate.
36         @param y The value for the Y coordinate.
37      */
38
39      public void setY(T y)
40      {
41         yCoordinate = y;
42      }
43
44      /**
45         The getX method returns the X coordinate.
46         @return The value of the X coordinate.
47      */
48
49      public T getX()
50      {
51         return xCoordinate;
52      }
53
54      /**
55         The getY method returns the Y coordinate.
56         @return The value of the Y coordinate.
57      */
58
59      public T getY()
60      {
61         return yCoordinate;
62      }
63   }
```

The only difference between this `Point` class and the one we studied earlier is the following type parameter notation in line 7:

```
public class Point<T extends Number>
```

The notation `<T extends Number>` defines a type parameter `T`, which is constrained with `Number` as its upper bound. In other words, only `Number` or a subclass of `Number` may be passed as a type argument to this parameter. The following statements will compile without error because each of the type arguments being used is a subclass of `Number`:

```
Point<Integer> iPoint = new Point<>(1, 2);
Point<Double> dPoint = new Point<>(1.5, 2.5);
Point<Long> lPoint = new Point<>(10L, 20L);
Point<Float> fPoint = new Point<>(7.9f, 9.9f);
```

These statements, however, will cause an error at compile time because the type arguments `String` and `Character` are not subclasses of `Number`:

```
// These statements will cause an error.
Point<String> iPoint = new Point<>("1", "2");
Point<Character> dPoint = new Point<>('1', '2');
```

You can also use the `super` key word in a generic class, to constrain a type parameter to a lower bound. For example, suppose we had constrained the `Point` class's type parameter as follows:

```
public class Point<T super Double>
```

This specifies that the type argument must be `Double`, or any superclass of `Double`. Otherwise, an error will occur at compile time.

**Checkpoint**

17.16 Why would you want to constrain the type parameter in a generic class?

17.17 Can you constrain a generic class's type parameter with an upper bound? With a lower bound?

## 17.6 Inheritance and Generic Classes

**CONCEPT: A generic class can be a superclass, and it can extend other classes.**

VideoNote
Inheritance and Generic classes

Inheritance can be used with generic classes. For example, Code Listing 17-8 shows the `Point3D` class, which extends the `Point` class. This class inherits the `Point` class's `xCoordinate` and `yCoordinate` fields, and has its own field for a *Z* coordinate, named `zCoordinate`. (This version of the class is stored in the source code folder *Chapter 17\Point Class Version 2*.)

**Code Listing 17-8** (`Point3D.java`)

```
/**
   The Point3D class holds a Z coordinate. The data type
   of the coordinate is generic.
*/

public class Point3D<T extends Number> extends Point<T>
{
   private T zCoordinate;   // The z coordinate

   /**
      Constructor
      @param x The X coordinate.
      @param y The Y coordinate.
      @param z The Z coordinate.
   */

   public Point3D(T x, T y, T z)
   {
```

```
19          // Call the Point class constructor.
20          super(x, y);
21  
22          // Assign the Z coordinate.
23          zCoordinate = z;
24       }
25  
26       /**
27          The setZ method sets the Z coordinate.
28          @param z The value for the Z coordinate.
29       */
30  
31       public void setZ(T z)
32       {
33          zCoordinate = z;
34       }
35  
36       /**
37          The getZ method returns the Z coordinate.
38          @return The value of the Z coordinate.
39       */
40  
41       public T getZ()
42       {
43          return zCoordinate;
44       }
45    }
```

Take a closer look at the class header in line 6:

```
public class Point3D<T extends Number> extends Point<T>
```

The first part of this statement reads as follows:

```
public class Point3D<T extends Number>
```

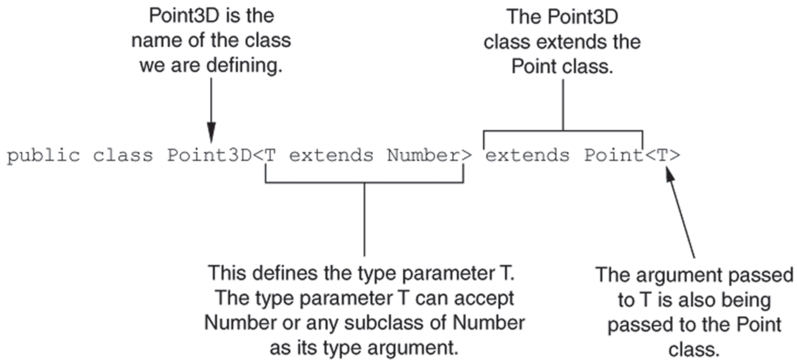
This indicates that the class we are defining is `Point3D`, and the type parameter is `T`. The notation `<T extends Number>` is used to define the type parameter, so any type argument passed to `T` must be `Number` or a subclass of `Number`. The next part of the statement is as follows:

```
extends Point<T>
```

This indicates that the class we are defining (`Point3D`) extends the `Point` class, with `T` passed as a type argument to the `Point` class. Figure 17-4 illustrates the parts of this code.

The program in Code Listing 17-9 demonstrates the class. (This program is stored in the source code folder *Chapter 17\Point Class Version 2*.)

**Figure 17-4** Generic subclass header


Point3D is the name of the class we are defining.
The Point3D class extends the Point class.
public class Point3D<T extends Number> extends Point<T>
This defines the type parameter T. The type parameter T can accept Number or any subclass of Number as its type argument.
The argument passed to T is also being passed to the Point class.

**Code Listing 17-9** (`TestPoint3D.java`)

```java
/**
   This program demonstrates the Point3D class.
*/

public class TestPoint3D
{
   public static void main(String[] args)
   {
      // Create a Point3D object that can hold Integers.
      Point3D<Integer> point = new Point3D<>(10, 20, 30);

      // Display the coordinates stored in point.
      System.out.println("Here are the values in point.");
      System.out.println("X Coordinate: " + point.getX());
      System.out.println("Y Coordinate: " + point.getY());
      System.out.println("Z Coordinate: " + point.getZ());
   }
}
```

**Program Output**

```
Here are the values in point.
X Coordinate: 10
Y Coordinate: 20
Z Coordinate: 30
```

The "is-a" relationship is in effect between the `Point3D` class and the `Point` class. A `Point3D` object *is a* `Point` object. So, we can assign a `Point3D` object to a `Point` reference variable, or pass a `Point3D` object to a method that accepts `Point` objects. The program in Code Listing 17-10 demonstrates this. (This program is also stored in the source code folder *Chapter 17\Point Class Version 2*.)

**Code Listing 17-10** (PassPoint3D.java)

```java
/**
   This program passes a Point3D object to a method
   with a Point parameter variable.
*/

public class PassPoint3D
{
   public static void main(String[] args)
   {
      // Create a Point3D object that can hold Integers.
      Point3D<Integer> point = new Point3D<>(10, 20, 30);

      // Display the X and Ycoordinates stored in point.
      System.out.println("Here are the X and Y coordinates.");
      printPoint(point);
   }

   /**
      The printPoint method prints a Point object's
      X and Y coordinates.
      @param point A Point object.
   */

   public static void printPoint(Point<?> point)
   {
      System.out.println("X Coordinate: " + point.getX());
      System.out.println("Y Coordinate: " + point.getY());
   }
}
```

**Program Output**

```
Here are the X and Y coordinates.
X Coordinate: 10
Y Coordinate: 20
```

The `Point3D` class is an example of a generic class that extends another generic class. Both generic and non-generic classes may be used together in an inheritance hierarchy, in any of the following ways:

- A generic class may extend another generic class
- A generic class may extend a non-generic class
- A non-generic class may extend a generic class

**Checkpoint**

17.18 Can a generic class be a superclass? Can it be a subclass?

17.19 A generic class extends another generic class. Can you assign a reference to an instance of the subclass to a superclass variable?

17.20 Can you mix generic and non-generic classes in a class hierarchy?

## 17.7 Defining Multiple Type Parameters

**CONCEPT: A generic class or method can have multiple type parameters, allowing it to accept multiple types as arguments.**

It is possible to define multiple type parameters in a generic class or method. Different type arguments can then be passed to each type parameter. Here is an example of a class that defines two type parameters:

```
public class MyClass<T, S>
{
   class code here...
}
```

In this example the type parameters are `T` and `S`. Inside the angled brackets the type parameters are listed, separated by commas. You can also apply constraints to the type parameters. Here is an example:

```
public class MyClass<T extends Number, S extends Date>
{
   class code here...
}
```

In this example the type parameters are `T` and `S`. Any type that is passed to `T` must be `Number` or a subclass of `Number`. Any type that is passed to `S` must be `Date` or a subclass of `Date`.

Code Listing 17-11 shows the `Pair` class, which uses two type parameters. The `Pair` class is designed to hold a pair of items. Each of the items can be of a different type.

**Code Listing 17-11** (`Pair.java`)

```
/**
   The Pair class demonstrates a generic class
   with two type parameters.
*/

public class Pair<T, S>
{
 8     private T first;        // The first item
 9     private S second;       // The second item
10
11     /**
12        Constructor
13        @param firstArg  Assigned to the first item.
14        @param secondArg Assigned to the second item.
15     */
16
17     public Pair(T firstArg, S secondArg)
18     {
19        first = firstArg;
20        second = secondArg;
21     }
22
23     /**
24        getFirst method
25        @return The first item in the pair.
26     */
27
28     public T getFirst()
29     {
30        return first;
31     }
32
33     /**
34        getSecond method
35        @return The second item in the pair.
36     */
37
38     public S getSecond()
39     {
40        return second;
41     }
42  }
```

Take a closer look at the class header in line 6:

```
public class Pair<T, S>
```

The notation `<T, S>` defines two type parameters, `T` and `S`. Notice that the first field is of the `T` type and the second field is of the `S` type. When we instantiate the class we must pass a type argument for each of these parameters. Code Listing 17-12 demonstrates the class.

**Code Listing 17-12** `(PairTest.java)`

```
 1  /**
 2     This program demonstrates the Pair class which
 3     has two type parameters.
 4  */
 5
 6  public class PairTest
 7  {
 8     public static void main(String[] args)
 9     {
10        // Create an Integer to hold an ID number.
11        Integer idNumber = new Integer(475);
12
13        // Create a String to hold a name.
14        String name = "Smith, Sally";
15
16        // Create a Pair object to hold the ID
17        // number and the name.
18        Pair<Integer, String> myPair =
19              new Pair<>(idNumber, name);
20
21        // Display the pair of items.
22        System.out.println("ID Number: " +
23                           myPair.getFirst());
24        System.out.println("Name: " +
25                           myPair.getSecond());
26     }
27  }
```

**Program Output**

```
ID Number: 475
Name: Smith, Sally
```

Line 11 creates an `Integer` object initialized with the value 475, referenced by the variable `idNumber`. Line 14 creates a `String` object referenced by the `name` variable. Lines 18 and 19 create a `Pair` object using the generic notation `<Integer, String>`. This passes `Integer` to the `T` parameter and `String` to the `S` parameter. This means that the `first` field will be an `Integer` and the `second` field will be a `String`.

The statement in lines 18 and 19 also pass `idNumber` and `name` to the `Pair` constructor, storing those items in the `myPair` object.

## 17.8 Generics and Interfaces

**CONCEPT: Interfaces, as well as classes, may be generic.**

So far, all of the generic types that we have examined have been classes. Interfaces can be generic too. For example, the Java API has an interface named `Comparable`, which is defined as follows:

```
public interface Comparable<T>
{
   int compareTo(T o);
}
```

This interface specifies a method named `compareTo`, which is used to compare the calling object with another object that is passed as an argument into the `o` parameter. Notice that the interface defines a type parameter, `T`. In the `compareTo` method header, `T` is the type of the `o` parameter.

Many of the classes in the Java API implement this interface, including `String` and the wrapper classes. You might recall from our discussion of the `String` class in Chapter 3 how the `compareTo` method works:

- If the calling object is less than the object passed to `o`, the method returns a negative number.
- If the calling object is equal to the object passed to `o`, the method returns 0.
- If the calling object is greater than the object passed to `o`, the method returns a positive number.

Suppose we are writing a class named `Tree`, and we want the `Tree` class to implement the `Comparable` interface. We would write the class in the following manner:

```
public class Tree implements Comparable<Tree>
{
   public int compareTo(Tree o)
   {
      Method code here...
   }

   Other class code here...
}
```

In the class header we are passing `Tree` as the type argument to the `Comparable` interface. So, the header for the `compareTo` method in the `Tree` class must be written as follows:

```
int compareTo(Tree o)
```

Any `Tree` object that calls the `compareTo` method will have to pass another `Tree` object as an argument to the method.

## Constraining a Type Parameter to a Type That Implements an Interface

It is common for a method to have a parameter variable that is an interface reference rather than a class reference. For example, suppose we wish to write a generic method named `greatest`, which will compare two objects and determine which one is the "greater" of the two. Of course, the notion of "greater" depends entirely on the objects that are being compared. If we are comparing two `LuxuryCar` objects, the one that is the most expensive might be considered the greater one. However, if we are comparing two `RaceCar` objects, the one that has the highest top speed might be considered the greater one. Any objects that are passed as arguments to our method will have to support a standard method of comparison so we can determine which is the greater.

The `Comparable` interface helps in a situation like this. For example, let's assume that the `LuxuryCar` class implements the `Comparable` interface. The class header would be written as follows:

```
public class LuxuryCar implements Comparable<LuxuryCar>
{
   Class code here...
}
```

If we have two `LuxuryCar` objects, `mercedes` and `jaguar`, we can compare the two objects with an expression such as:

```
mercedes.compareTo(jaguar)
```

This expression will return a negative number, 0, or a positive number, indicating that the `mercedes` object is less than, equal to, or greater than the `jaguar` object.

The way we should write our `greatest` method is to make sure that only objects that implement the `Comparable` interface are passed to it as arguments. That way, we can safely use one of the objects to call the `compareTo` method, and determine which object is the greatest. With generics, we could write the method as follows:

```
public static < T extends Comparable<T> > T greatest(T arg1, T arg2)
{
   if (arg1.compareTo(arg2) > 0)
      return arg1;
   else
      return arg2;
}
```

This method defines a type parameter `T` with the following notation:

```
< T extends Comparable<T> >
```

In generic notation, the key word `extends` also means "implements." This specifies that the type passed to `T` must implement the `Comparable` interface. The method has two parameter variables, `arg1` and `arg2`. Because these parameter variables are of type `T`, any objects passed to them must implement the `Comparable` interface. Notice that the method's return type is also `T`, meaning that the method will return a reference to an object of type `T`.

Code Listing 17-13 shows a program that uses a generic method to search an array for a value. Both the array and the search value must be of a type that implements the `Comparable` interface. This is because the method uses the `compareTo` method to determine whether an element contains the search value.

**Code Listing 17-13** (GenericSearchArray.java)

```
/**
   This program uses a generic method to sequentially
   search an array for a value.
*/

public class GenericSearchArray
{
   public static void main(String[] args)
   {
      int position;  // To hold a string's position in the array

      // Array of strings to search
      String[] names = { "Jack", "Kelly", "Beth",
                         "Chris", "Kenny", "Britainy" };

      // Search the array for Chris.
      position = sequentialSearch(names, "Chris");

      // Determine whether Chris was found.
      if (position == -1)
         System.out.println("Chris is not in the array. ");
      else
         System.out.println("Chris is at position " + position);
   }

   /**
      The sequentialSearch method searches an array for
      a value.
      @param array The array to search.
      @param value The value to search for.
      @return The subscript of the value if found in the
              array, otherwise -1.
   */

   public static < E extends Comparable<E> >
            int sequentialSearch(E[] array, E value)
   {
      int index;         // Loop control variable
      int position;      // Position the value is found at
      boolean found;     // Flag indicating search results
41
42       // Position 0 is the starting point of the search.
43       index = 0;
44
45       // Store the default values for position and found.
46       position = -1;
47       found = false;
48
49       // Search the array.
50       while (!found && index < array.length)
51       {
52          if (array[index].compareTo(value) == 0)
53          {
54             found = true;
55             position = index;
56          }
57          index++;
58       }
59       return position;
60    }
61 }
```

**Program Output**

```
Chris is at position 3
```

Take a closer look at the `sequentialSearch` method header, which appears in lines 35 and 36:

```
public static < E extends Comparable<E> >
             int sequentialSearch(E[] array, E value)
```

This method header defines a type parameter, `E`, which is constrained to types that implement the `Comparable` interface. The method's first parameter variable, `array`, is an array of type `E`. The second parameter variable, `value`, is a reference to an object of type `E`. If `value` is found in the array, the method returns the subscript of the element containing it. If `value` is not found in the array, the method returns –1.

**Checkpoint**

17.21 Can interfaces be defined as generic?

17.22 Assume that you must define the type parameter `T` in a generic method. How would you constrain the type parameter `T` so it only accepts type arguments that implement the `Comparable` interface?

## In the Spotlight:

### Generic Subsets

Consider the task of generating a random subset of a set of some type. You might need to do this if you are writing a game of chance or if you are simulating certain types of real-world phenomena.

Let us suppose the set whose random subset we are to generate is in the form of a generic list,

```
List<T> argList
```

where `T` is a type parameter. There are different strategies we can use to pick a random subset: Here is one approach.

If the set has size $N$, it will have $2^N$ subsets. Each subset will correspond to a bit string of length $N$; and in turn, each bit string will correspond to a binary number in the range $0..2^N - 1$. For example, if the set is $S = \{A, B, C\}$, then $N = 3$ and each of the eight subsets of $S$ corresponds to a bit string of length 3. This three-way correspondence between subsets, bit strings, and numbers is illustrated in the following table.

| **Subset** | $\{A, B, C\}$ | $\{A, B\}$ | $\{A, C\}$ | $\{A\}$ | $\{B, C\}$ | $\{B\}$ | $\{C\}$ | $\{\ \}$ |
|---|---|---|---|---|---|---|---|---|
| Bit string | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
| Number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Briefly, each element of the original set is assigned a position. A bit string has a 1 in that position whenever the corresponding subset includes the element at that position. For a set of size $N$, we can use the `nextInt()` method of the `java.util.Random` class to select a random integer $k$ in the range $0..2^N - 1$. We can next convert $k$ to the binary string that represents it and then map this binary string to its corresponding subset. For example, if the random number selected is 5, then the bit string is 101, and the selected random subset is $\{A, C\}$. In this way, we can select random subsets of a set of size $N$ by selecting random integers in the range $0..2^N - 1$.

There are two static methods of the `Integer` class that will prove useful. The first is as follows:

```
int rotateLeft(int i, int N)
```

Calling this method with a value of 1 for the first parameter $i$ and a value of $N$ less than 32 will return $2^N$, the number of subsets in a set of size $N$. This is shown in Line 41 of the following code listing. To convert an integer to the corresponding bit string, we use the method

```
String toBinaryString(int i);
```

as shown in line 47. Contrary to what you might expect, the length of the string returned, `subsetStr`, may not be the same as the size of the `argList`, the input set. This is because leading zeroes are omitted. For example, the bit string corresponding to $\{B, C\}$ will be 11 rather than 011. To compensate, the code that maps `1` bits to list elements must shift the index into `argList` by the number of zero bits omitted. This code appears in lines 50–55.

The following code listing demonstrates a generic class with a generic method for constructing a random subset. The `main` method exercises the method by calling the method three times, passing it the same list of strings on each call. As the output of the program shows, each call to the method is likely to return a different subset.

**Code Listing 17-14** (GenericRandomSubset.java)

```
import java.util.*;

/**
  This program shows how to generate a random subset of a set S
  when S is given as a generic list.
*/

public class GenericRandomSubset <T>
{
   public static void main(String[] args)
   {
      // Create a random subsets object specialized for strings
      Subsets<String> strSubsets = new Subsets<>();

      String [] people = {"Moe", "Curly", "Larry", "Bobby", "Stinky"};

      // Transform the array into a list
      List<String> peopleList = Arrays.asList(people);
      System.out.println(strSubsets.getRandomSubset(peopleList));
      System.out.println(strSubsets.getRandomSubset(peopleList));
      System.out.println(strSubsets.getRandomSubset(peopleList));
   }
}

class Subsets<T>
{
   Random rand = new Random();

   /**
     This method computes a random subset of an input set.
     @param argList: represents the input set
     @return a list representing a random subset of a set
   */

   List<T> getRandomSubset(List<T> argList)
   {
      List<T> resultList = new ArrayList<T>();
      int argListSize = argList.size();

      // Compute the size of the power set of the input set.
      int powerSetSize = Integer.rotateLeft(1,argListSize);

      // Select a random integer in the range 0 to powerSetSize-1.
      int randomInt = rand.nextInt(powerSetSize);

      // Convert the random integer to a subset
      String subsetStr = Integer.toBinaryString(randomInt);

      // Interpret each bit in the string as an element
      int length = subsetStr.length();
      for (int k = 0; k < length; k++)
      {
         if (subsetStr.charAt(k) == '1')
            resultList.add(argList.get(k + argList.size()-length));
      }
      return resultList;
   }
}
```

**Program Output**

```
[Moe, Curly, Larry, Bobby]
[Moe, Curly, Bobby]
[Bobby, Stinky]
```

## 17.9 Erasure

**CONCEPT: When the Java compiler processes a generic class or method, it erases the generic notation and substitutes an actual type for each type parameter.**

When the Java compiler encounters generic code, it uses a process known as *erasure* to compile the code. The process is called erasure because the compiler erases the generic notation and substitutes an actual type for each type parameter. Here's how erasure works: When the compiler encounters a generic class, interface, or method with an unbound type parameter, such as `<T>` or `<E>`, it replaces all occurrences of the type parameter with `Object`. For example, the following class:

```
public class Point<T>
{
   private T xCoordinate;
   private T yCoordinate;

   public Point(T x, T y)
   { code... }

   public void setX(T x)
   { code... }

   public void setY(T y)
   { code... }

   public T getX()
   { code... }
   public T getY()
   { code... }
}
```

would be converted by the process of erasure to this code:

```
public class Point
{
   private Object xCoordinate;
   private Object yCoordinate;

   public Point(Object x, Object y)
   { code... }

   public void setX(Object x)
   { code... }

   public void setY(Object y)
   { code... }

   public Object getX()
   { code... }

   public Object getY()
   { code... }
}
```

And, the following generic method:

```
   public static <E> void displayArray(E[] array)
   {
      for (E element : array)
         System.out.println(element);
   }
```

would be converted to:

```
public static void displayArray(Object[] array)
{
   for (Object element : array)
      System.out.println(element);
}
```

When the compiler encounters a class, interface, or method with a bound type parameter, such as `<T extends Number>` or `<E extends Comparable>`, it replaces all occurrences of the type parameter with the bound that is applied to the parameter. For example, the following class:

```
public class Point<T extends Number>
{
   private T xCoordinate;
   private T yCoordinate;
   public Point(T x, T y)
   { code... }

   public void setX(T x)
   { code... }

   public void setY(T y)
   { code... }

   public T getX()
   { code... }

   public T getY()
   { code... }
}
```

would be converted by the process of erasure to this code:

```
public class Point
{
   private Number xCoordinate;
   private Number yCoordinate;

   public Point(Number x, Number y)
   { code... }

   public void setX(Number x)
   { code... }

   public void setY(Number y)
   { code... }

   public Number getX()
   { code... }

   public Number getY()
   { code... }
}
```

And, the following generic method:

```
public static <E extends Comparable<E>>
   int sequentialSearch(E[] array, E value)
{
   method code here...
}
```

would be converted to:

```
public static
      int sequentialSearch(Comparable[] array, Comparable value)
{
   method code here...
}
```

As you can see, generic notation exists only in the source code that is written by the programmer. Once the compiler confirms that a generic type is being used safely, it converts it into a raw type, which is simply a non-generic type. At the bytecode level, only raw types exist.

When working with generic types, the compiler also inserts cast operators, where necessary, in the code that uses the generic type. For example, suppose we use the following code to create an instance of the `Point` class:

```
Integer x = new Integer(1);
Integer y = new Integer(2);
Point<Integer> myPoint = new Point<>(x, y);
```

Then, later in the same program we write the following statement:

```
Integer tempX = myPoint.getX();
```

Because the translated `getX` method returns an `Object` reference, the compiler will modify this statement to read the following:

```
Integer tempX = (Integer)myPoint.getX();
```

Generics were introduced in Java 5. Older versions of Java did not support generics, so programmers could only write raw classes. The process of erasure ensures that newer code utilizing generics can coexist with older code that was written before Java 5.

**Checkpoint**

17.23 During the process of erasure, when the compiler encounters a generic class, interface, or method with an unbound type parameter, such as `<T>` or `<E>`, what does it replace the type parameter with?

17.24 During the process of erasure, when the compiler encounters a class, interface, or method with a bound type parameter, such as `<T extends Number>` or `<E extends Comparable>`, what does it replace the type parameter with?

17.25 Do generic types still exist at the bytecode level?

## 17.10 Restrictions on the Use of Generic Types

**CONCEPT: Type parameters cannot be used in every way that a real type can be used.**

Because of the erasure process, there are restrictions on how and when generic types and type parameters can be used. In this section we will examine some of those restrictions.

**You cannot create an instance of a type parameter.**

Let's say that a generic class has a type parameter named `T`. In the class you cannot create an instance of `T`. For example, the following code will not compile:

```
1 public class MyClass<T>
2 {
3    public MyClass()
4    {
5       // The following statement causes an ERROR.
6       T myObject = new T();
7    }
8 }
```

In this code the statement in line 6 causes an error at compile time. The expression `new T()` implies that `T` is the name of a constructor, but it is not.

**You cannot create an array of generic class objects.**

The Java compiler will not allow you to create an array of objects that are instances of a generic type. For example, the following statement will not compile:

```
ArrayList<String>[] a = new ArrayList<>[100];
```

At runtime, arrays in Java carry internal information about the data type of their elements. Because of erasure, an array of generic class instances would not have the necessary runtime type information, so the compiler will not allow such arrays to be created.

**A generic class's type parameter cannot be the type of a static field, and cannot be referred to in a static method.**

You cannot use a type parameter as the type of a static field. For example, the following code will not compile:

```
public class MyClass<T>
{
   // The following statement will cause an ERROR.
   private static T value;
}
```

Regardless of the number of instances of `MyClass`, there is only one instance of the static variable `value`, and therefore `value` can have only one type. The compiler also prohibits a static method from referring to a class's type parameter. The following code is illegal:

```
public class MyClass<T>
{
   public static void doSomething()
   {
      // The following statement causes an ERROR.
      T myValue;
   }
}
```

**You cannot make an exception class generic.**

Let's imagine what would have to take place in order for a Java application to use generic exception classes. First, you would have to be able to use generic type notation in a `catch` clause. Then, when a generic exception was thrown, the system would have to check the type argument used to create the exception class to see if it matches the type specified in the catch clause. This is not possible because at runtime, generic type information has already been erased by the compiler. Because the compiler cannot know at compile time where an exception will come from, or what type argument will be used to create it, you are not allowed to create a generic exception class. Specifically, you are not allowed to create a generic subclass of `Throwable`. You are also not allowed to use generic type notation in a `catch` clause.

## 17.11 Common Errors to Avoid

- **Forgetting to pass a type argument when instantiating a generic class.** When you instantiate a generic class without passing a type argument, you are using the class as a raw type. When you do this, you give up the benefits of type-safety that you normally have with a generic class.
- **Using a primitive type as a type argument.** Generic classes and methods can accept only reference types as type arguments.
- **Misusing inheritance when passing an instance of a generic class as an argument to a method.** If a method's parameter variable is declared as the `Point<Number>` type, you cannot pass an instance of `Point<Integer>` to the parameter. Even though `Integer` is a subclass of `Number`, `Point<Integer>` is not a subclass of `Point<Number>`. When a parameter variable is declared as `Point<Number>`, the only arguments that can be passed to it are instances of `Point<Number>`.
- **Being too type-restrictive when using a generic class to declare a parameter variable.** Suppose in a method you declare a parameter variable as `Point<Integer> myPoint`. The only arguments that can be passed to this parameter are those of the `Point<Integer>` type. If you try to pass a `Point<Double>` object, an error will occur. By declaring the parameter as `Point<?> myPoint`, you will be able to pass any `Point` object to the method. By declaring the parameter variable as `Point<? extends Number> myPoint`, you will be able to pass any `Point` object with a type argument that is `Number` or a subclass of `Number`.
- **Trying to use the implements key word to constrain a type parameter to types that implement an interface.** You do not use the `implements` key word in generic type notation. To constrain a type parameter to types that implement a particular interface, you use the `extends` key word. In generic notation, the `extends` key word means "extends" or "implements."
- **Forgetting to define a type parameter in a generic method.** If a generic method has its own type parameter, you must write a definition for the type parameter in the method header.
- **Defining a type parameter in the wrong place in a generic method.** Type parameter definitions in a generic method must appear just before the method's return type.

# Review Questions and Exercises

## Multiple Choice and True/False

1. When you create an instance of a generic class, what types can you pass as arguments to the class's type parameter?
   a. primitive types only
   b. reference types only
   c. interface types only
   d. primitive, reference, and interface types

2. In a generic method, a type parameter is defined _______.
   a. inside the parentheses, along with the method's parameter variables
   b. after the method's return type, and before the method's name
   c. before the method's return type
   d. inside the body of the method, but before any local variables are declared

3. Look at the following method header:

   `void displayPoint(Point<Number> myPoint)`

   Which of the following objects would we be allowed to pass as an argument to the `displayPoint` method? (Select all that apply.)
   a. `Point<Number> p;`
   b. `Point<Integer> p;`
   c. `Point <Double> p;`
   d. `Point <String> p;`

4. Look at the following method header:

   `void displayPoint(Point<?> myPoint)`

   Which of the following objects would we be allowed to pass as an argument to the `displayPoint` method? (Select all that apply.)
   a. `Point<Number> p;`
   b. `Point<Integer> p;`
   c. `Point <Double> p;`
   d. `Point <String> p;`

5. Look at the following method header:

   `void displayPoint(Point<? extends Number> myPoint)`

   Which of the following objects would we be allowed to pass as an argument to the `displayPoint` method? (Select all that apply.)
   a. `Point<Number> p;`
   b. `Point<Integer> p;`
   c. `Point <Double> p;`
   d. `Point <String> p;`

6. Look at the following method header:

   `void displayPoint(Point<? super Double> myPoint)`

   Which of the following objects would we be allowed to pass as an argument to the `displayPoint` method? (Select all that apply.)

   a. `Point<Number> p;`
   b. `Point<Integer> p;`
   c. `Point <Double> p;`
   d. `Point <String> p;`

7. In the generic type notation `<T extends Number>` to what type of bound is `T` constrained?

   a. an upper bound
   b. a lower bound
   c. both an upper bound and lower bound
   d. `T` is not constrained to a bound

8. In the generic type notation `<T super Integer>` to what type of bound is `T` constrained?

   a. an upper bound
   b. a lower bound
   c. both an upper bound and lower bound
   d. `T` is not constrained to a bound

9. Which of the following generic type notations uses a wildcard?

   a. `Point<W>`
   b. `Point<T extends Number>`
   c. `Point<T super Integer>`
   d. `Point<?>`

10. The process used by the Java compiler to remove generic notation and substitute an actual type for type parameters is known as _______.

    a. erasure
    b. removal
    c. substitution
    d. masking

11. **True or False:** It is better to discover an error at runtime than at compile time.
12. **True or False:** It is possible to instantiate a generic class without specifying a type argument.
13. **True or False:** Type parameters must be single character identifiers, written in uppercase.
14. **True or False:** You can constrain a type parameter to both an upper bound and lower bound.
15. **True or False:** A generic class can extend a non-generic class.

16. **True or False:** You cannot create an array of generic class objects.
17. **True or False:** A generic class's type parameter can be the type of a static field.
18. **True or False:** An exception class cannot be generic.

## Find the Error

1. `ArrayList myList = new ArrayList<String>();`
2. Assume the following is a method header in a class:

```
public < T implements Comparable<T> > T greatest(T arg1, T arg2)
```

3.

```
public class MyClass<T>
{
   public static void displayValue(T value)
   {
      System.out.println(value);
   }
}
```

4.

```
public class Point<T extends Number super Integer>
{
   Class code . . .
}
```

## Algorithm Workbench

1. Assume there is a class named `Customer`. Write a statement that creates an `ArrayList` that can hold only `Customer` objects.
2. Assume `names` references an object of the `ArrayList<String>` class. Write a statement that adds the string `"William Jefferson"` to the `names` object.
3. Write the header for a generic class named `MyType`. The class should have one type parameter. The type parameter's upper bound should be the `String` class.
4. Write the header for a generic class named `MyType`. The class should have one type parameter. The type parameter's lower bound should be the `Integer` class.
5. Modify the following `max` method so its arguments can be of any type that implements the `Comparable` interface. The method's return type should be the same as the type of its parameter variables.

```
public static int max(int a, int b)
{
   if (a > b)
      return a;
   else
      return b;
}
```

6. Write the header for a generic class named `MyType`. The class should have two type parameters.

7. Write the header for a generic class named `MyType`. The class should have two type parameters. The first type parameter's upper bound should be the `Number` class. The second type parameter's upper bound should be the `String` class.

**Short Answer**

1. When a generic class is used in a program, when does type checking take place?
2. Look at the following method header:

   `public < T extends Comparable<T> > T least(T arg1, T arg2)`

   What constraint is placed on the type parameter `T`?
3. What is erasure?
4. Do generic types exist at the bytecode level?
5. When the compiler encounters a generic class, interface, or method with an unbound type parameter, such as `<T>` or `<E>`, it replaces all occurrences of the type parameter with what type?
6. When the compiler encounters a class, interface, or method with a bound type parameter, such as `<T extends Number>` or `<E extends Comparable>`, it replaces all occurrences of the type parameter with what type?

## Programming Challenges

### 1. `MyList` Class

Write a generic class named `MyList`, with a type parameter `T`. The type parameter `T` should be constrained to an upper bound: the `Number` class. The class should have as a field an `ArrayList` of `T`. Write a public method named `add`, which accepts a parameter of type `T`. When an argument is passed to the method, it is added to the `ArrayList`. Write two other methods, `largest` and `smallest`, which return the largest and smallest values in the `ArrayList`.

### 2. `MyList` Modification

Modify the `MyList` class that you wrote for Programming Challenge 1 so the type parameter `T` should accept any type that implements the `Comparable` interface. Test the class in a program that creates one instance of `MyList` to store `Integer`s, and another instance to store `String`s.

### 3. `PointList` Class

Write a class named `PointList` that keeps a list of `Point` objects in an `ArrayList`. The `PointList` class should accept any object that is an instance of the `Point` class, or a subclass of `Point`. Demonstrate the class in an application.

### 4. Generic Insertion Sort

In Programming Challenge 1 of Chapter 16 you wrote an `ObjectInsertionSorter` class that can sort `Comparable` objects. Modify the sorting method so it is a generic method. The method should use generic notation to accept an array of any objects that implement the `Comparable` interface. Demonstrate the class in an application.

### 5. Generic Binary Search

In Programming Challenge 3 of Chapter 16 you wrote an `ObjectBinarySearcher` class. Modify the search method so it is a generic method. The method should use generic notation to accept an array of any objects that implement the `Comparable` interface. Demonstrate the class in an application.

VideoNote
Highest and Lowest Elements

### 6. Highest and Lowest Elements

Write a generic class with a type parameter constrained to any type that implements `Comparable`. The constructor should accept an array of such objects. The class should have methods that return the highest and lowest values in the array. Demonstrate the class in an application.

### 7. Number Analyzer

Write a generic class with a type parameter constrained to the `Number` class or any subclass of `Number`. The constructor should accept an array of such objects. The class should have methods that return the highest and lowest values in the array, the total of the elements, and the average value of all the elements. Demonstrate the class in an application.

### 8. Generic Random Permutation

Write a generic class with a generic method that returns a random permutation of a list passed to it as a parameter. Demonstrate the working of the method in an application that displays random permutations of lists of strings and of lists of numbers.

CHAPTER 18

# Collections and the Stream API

**TOPICS**

18.1 Introduction to the Java Collections Framework
18.2 Lists
18.3 Sets
18.4 Maps
18.5 The Collections Class
18.6 The Stream API
18.7 Common Errors to Avoid

## 18.1 Introduction to the Java Collections Framework

**CONCEPT: The Java API provides numerous interfaces and classes that allow you to create collections, which are objects that act as containers for groups of other objects.**

It is common for an application to create a group of similar objects and work with that group of objects throughout the application's execution. Suppose, a teacher uses an application to manage student grades. That application might create several `Student` objects, with each object holding information about a particular student. Another example would be a payroll application that creates a group of `Employee` objects, with each object holding information about a specific employee.

Working with groups of objects is so common in programming that the Java API provides a set of interfaces and classes specifically for this purpose. These interfaces and classes, which are in the `java.util` package, are known as the *Java Collections Framework (JCF)*. You can use these interfaces and classes to create *collections*, which are objects that act as *containers* of other objects. You can use a collection to store a group of objects. Each object that is stored in a collection is referred to as an *element*. Collections allow you to add, retrieve, and delete objects.

### Lists, Sets, and Maps

There are three general types of collections: *lists*, *sets*, and *maps*. Each of these is described here:

- List—A *list* is an *ordered collection* that orders its elements by position. The *position* of an element *x* within the list, also called its *index*, is the number of elements that precede $x$ within the list. A list permits duplicate elements: two objects that are equal can be added to the list. The list distinguishes equal objects by their index.

- Set—A *set* in an *unordered collection* of objects that does not permit duplicates. Unlike a list, a set has no concept of position for its elements. Implementations of sets are usually optimized for searching so you can quickly find a specific stored value.
- Map—A *map* is a collection that stores every object $x$ according to a property of $x$ called its *key*. This means that each element in a map is a *key-value pair*. The *key* is a property of the object and the *value* is the object itself. A map allows a value to be quickly retrieved based on its key.

## Java Functional Interfaces

The `java.util.function` package defines a number of functional interfaces that are referred to by methods defined in JCF interfaces and classes. In practice, the functional interfaces are used to declare the types of lambda expressions to be passed as a parameter to a collection class method. Knowing the naming system used to name these functional interfaces will help you to better understand the JCF methods that refer to them. The naming system uses the words *supplier*, *consumer*, *predicate, operator*, and *function* as root words, and employs a set of prefixes that includes *binary*, *bi*, *unary*, and the names of primitive types to modify the meaning of the root words.

A *function* takes at least one parameter and returns a value. The generic interface `Function<T, R>` takes a parameter of type `T` and returns a value of type `R`. A *bi-function* takes two parameters and returns a value: the generic interface `BiFunction<T, U, R>` takes two parameters of type `T` and `U`, in that order, and returns a value of type `R`.

It is useful to have a short hand notation for functional types. We will use the notation $T \rightarrow R$ for the type `Function<T, R>` and $(T, U) \rightarrow R$ for the type `BiFunction<T, U, R>`. Similar shorthand notation will be used for other functional types.

There are also functional interfaces that involve the primitive types `int`, `long`, and `double`. For example, the interface `IntFunction<R>` is the type `int` $\rightarrow R$, the interface `LongFunction<R>` is the type `long` $\rightarrow R$, and `IntToLongFunction` is the type `int` → `long`.

An *operator* is a function whose parameters and return value have the same type. A *binary operator* is an operator that takes two parameters, and a *unary operator* is an operator that takes a single parameter. Thus `BinaryOperator<T>` has type $(T, T) \rightarrow T$, the interface `UnaryOperator<T>` has type $T \rightarrow T$, and `LongUnaryOperator` has type `long` → `long`.

A *consumer* takes at least one parameter and returns no result. The interface `Consumer<T>` has type $T \rightarrow$ `void`, while the interface `BiConsumer<T, U>` has type $(T, U) \rightarrow$ `void`.

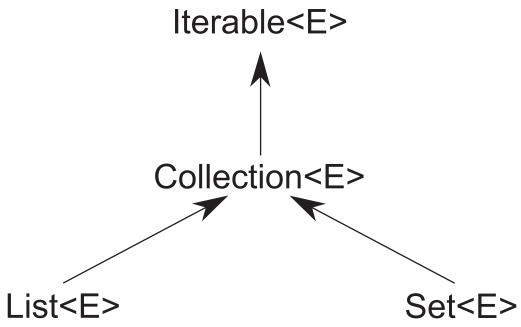
A *supplier* is the opposite of a consumer: it takes no parameters and returns a value. The interface `Supplier<T>` has type $(\ ) \rightarrow T$ and `IntSupplier` has type $(\ ) \rightarrow$ `int`.

A *predicate* is a function that returns a boolean value. Thus `Predicate<T>` has type $T \rightarrow$ `boolean`, while `BiPredicate<T, U>` has type $(T, U) \rightarrow$ `boolean`.

## JCF Interfaces

The Java Collection Framework defines interfaces and classes to facilitate the creation and use of collections. The classes and interfaces are generic so they can be used with different element types. Four of these interfaces are shown in Figure 18-1.

**Figure 18-1** The main JCF interfaces



## Iterating a Collection

Figure 18-1 shows that lists and sets are special types of collections, and that every collection is iterable. *Iteration* is the successive retrieval of elements from a collection until all elements have been examined. A collection that provides methods for iteration is said to be *iterable*. The JCF `Iterable<E>` interface defines methods for iterating a collection that stores elements of type `E`. The `Collection` interface extends `Iterable` and adds many methods for working with collections. The `List` interface is implemented by all collections that order their elements by position. It defines methods for adding, removing, and retrieving elements by index. Finally, the `Set` interface serves as the interface for all unordered collections. The `Set` interface does not add any methods to those that it inherits from `Collection`.

There is also a `Map<K,V>` interface that defines methods for a map that stores key-value pairs with `K` as the type of key and `V` as the type of value. This interface does not extend `Collection`, and will be covered later.

## The `Iterable` Interface and the `forEach` method

The `Iterable<E>` interface provides two methods that can be used to iterate through a collection. The first of these is the method:

```
void forEach(Consumer<? super E> action)
```

The `action` parameter must be an object that implements the `Consumer<? Super E>` interface. In practice, this will usually be a lambda expression that takes a single parameter and returns no value. Moreover, the type of the parameter must be a superclass of `E`, the collection's element type.

We can illustrate the use of `forEach()` using `ArrayList`, a class that you learned about in Chapter 7. The program in Code Listing 18-1 creates a list with four names, and then uses `forEach()` to invoke the consumer function defined by the lambda expression

```
x ->
{
    System.out.printf("%s %d\n", x, x.length());
}
```

on each element of the list. The effect is to print the name and length of each element in the list.

**Code Listing 18-1** (ForEachDemo.java)

```
import java.util.ArrayList;

/**
   This program demonstrates Iterable forEach().
*/

public class ForEachDemo
{
    public static void main(String [] args)
    {
        // Array of names.
        String [] names = {"Anna", "Bob", "Carlos", "Debby"};

        // Create list and add names.
        ArrayList<String> nameList = new ArrayList<>();
        for (String name : names)
        {
            nameList.add(name);
        }

        // Use forEach with lambda expression to print.
        nameList.forEach(
                x ->
                 {
                     System.out.printf("%s %d\n", x, x.length());
                 });
    }
}
```

**Program Output**

```
Anna 4
Bob 3
Carlos 6
Debby 5
```

## Iterators

Another method in the `Iterable<E>` is as follows:

```
Iterator<E> iterator()
```

Calling this method on a collection returns an iterator for that collection. An *iterator* for a collection *C* is an object that provides controlled iteration through the elements of *C*. Unlike the `forEach()` method, which applies an action to *all* elements of a collection, an iterator provides finer control, letting you call for the next element only when you need it, and allowing you to stop the iteration at any stage of the process. An iterator is defined by the `Iterator<E>` interface, whose methods are defined in Table 18-1.

**Table 18-1** Methods of the `Iterator<E>` interface

| | |
|---|---|
| `boolean hasNext()` | Returns `true` if there are elements that have not yet been returned by the iterator, and `false` otherwise. |
| `E next()` | Returns the next element from the collection, or throws `NoSuchElementException` if no more elements remain. |
| `void remove()` | Removes from the underlying collection the element that was returned by the last call to `next()`. You can only call this method once after each call to `next()`. |
| `void forEachRemaining(Consumer<? super E> action)` | Calls the specified action method for each element that has not yet been returned by the iterator. |

You get an iterator by calling the `iterator()` method on a collection. This method will take a snapshot of the elements in the collection at the time of the call and return an iterator that is ready to provide access to those elements. You typically use the iterator in a loop, calling `hasNext()` to see if there are more elements yet to be examined, and calling `next()` to fetch the next available element. The `remove()` method is optional, so it is not supported by all iterators. If supported, it will remove the element returned by the last call to `next()` from the underlying collection. The `forEachRemaining()` method applies an action to every element that has not yet been returned by the iterator.

The program in Code Listing 18-2 is a variation of the one in Code Listing 18-1. It prints names and lengths until `"Bob"` has been printed, then it prints the name without the length for each remaining list element.

To make the code more readable, the program defines the lambda expression in a separate statement:

```
Consumer<String> action = x ->
              {
                  System.out.printf("%s\n", x);
              };
```

Notice that the functional interface must be specialized to the element type of the collection, which is `String`. The iterator is created like this:

```
Iterator<String> iter = nameList.iterator();
```

The call `iter.hasNext()` checks to if there are still elements to be returned by the iterator, and the call to `iter.next()` returns the next available element.

**Code Listing 18-2** `(IteratorDemo.java)`

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.function.Consumer;
```

```

/**
  This program demonstrates Iterators.
*/

public class IteratorDemo
{
    public static void main(String [] args)
    {
        // Array of names.
        String [] names = {"Anna", "Bob", "Carlos", "Debby"};

        // Create list and add names.
        ArrayList<String> nameList = new ArrayList<>();
        for (String name : names)
        {
            nameList.add(name);
        }

        // Define an action for the "remaining" elements.
        Consumer<String> action = x ->
                 {
                     System.out.printf("%s\n", x);
                 };

        // Get the iterator to the list.
        Iterator<String> iter = nameList.iterator();

        // Process list elements with the iterator.
        while (iter.hasNext())
        {
            String name = iter.next();
            System.out.printf("%s %d\n", name, name.length());
            if (name.equals("Bob"))
            {
                // Act differently for names after "Bob".
                iter.forEachRemaining(action);
            }
        }
    }
}
```

**Program Output**

```
Anna 4
Bob 3
Carlos
Debby
```

You can think of an iterator as having a list of all elements in the collection and a cursor that is initially positioned just before the first element of that list. Every call to `next()` returns the element at the cursor position and moves the cursor past the element returned. The `hasNext()` method returns false when the cursor gets past the last element in the list.

**NOTE:** The `next()` method throws `NoSuchElementException` if it is called when there are no elements left to return.

## The Enhanced `For` Loop

In Chapter 7, you learned that the enhanced `for` loop can be used with arrays and `ArrayList` collections. In fact, the enhanced for loop will work with any object of any class that implements the `Iterable` interface. In particular, it can be used with any collection.

## The `Collection` Interface

The `Collection<E>` interface defines methods for working with collections. Some of these methods, with brief explanations of their use, are shown in Table 18-2.

**Table 18-2** Methods of the `Collection<E>` interface

| | |
|---|---|
| `boolean add(E e)` | Adds the object referenced by `e` to this collection. Some collections restrict the objects that can be added as elements. This method returns `true` if the object was added to the collection. It returns `false` if the collection did not allow the object to be added. This would be the case if the collection is a set and the `add()` is attempting to add a duplicate. |
| `boolean addAll(Collection<? extends E> c)` | The parameter `c` is another `Collection` object. This method adds all of the elements of `c` to this collection. The method returns `true` if the elements of `c` were added to this collection. It returns `false` otherwise. |
| `void clear()` | Removes all of the elements in this collection. |
| `boolean contains(Object o)` | Returns `true` if this collection contains the specified object, or `false` otherwise. |
| `boolean containsAll(Collection<?> c)` | Returns `true` if this collection contains all of the elements in the collection `c`, and returns `false` otherwise. |
| `boolean equals(Object o)` | Returns `true` if this collection is equal to the object `o`, and `false` otherwise. |
| `boolean isEmpty()` | Returns `true` if this collection contains no elements, and `false` otherwise. |
| `boolean remove(Object o)` | Removes a single instance of the object `o` from this collection if found, and returns `true`. If `o` is not found, the method returns `false`. |

*(table continues next page)*

**Table 18-2** Methods of the `Collection<E>` interface (continued)

| | |
|---|---|
| `boolean removeAll(Collection<?> c)` | Removes every element in the collection `c` from this collection. Returns `true` if at least one element was removed from this collection, and `false` otherwise. |
| `boolean removeIf(Predicate<? super E> filter)` | Removes all of the elements of this collection that satisfy the given predicate. Returns `true` if at least one element was removed from this collection, and `false` otherwise. |
| `boolean retainAll(Collection<?> c)` | The parameter `c` is another collection. This collection will retain all of its elements that are also elements of the collection `c`. All other elements in this collection are removed. Returns `true` if at least one element was removed. |
| `int size()` | Returns the number of elements in this collection. |
| `Stream<E> stream()` | Returns a sequential stream with this collection as its source. Streams are discussed later in this chapter. |
| `Object[] toArray()` | This method converts the collection into an array of `Object` and returns the array. |
| `<T> T[] toArray(T[] a)` | The parameter to this method is an array of some reference type `T`. The elements of this collection are stored as objects of type `T` in the array `a` and the array `a` is returned. If this collection contains more elements than the array `a`, then a new array is created to hold the elements of this collection and the new array is returned. If this collection holds fewer elements than the array `a`, then the elements of this collection are stored in `a` and the element of `a` that immediately follows the last collection element is set to `null`. |

The `removeIf` method uses a predicate as a filter. This method filters out all elements that satisfy the collection, leaving in the collection only those elements on which the predicate returns a value of false. Code Listing 18-3 uses this method to remove from a list of names all names whose length is at most 4. First, we define a lambda expression to be used as the filter:

```
Predicate<String> filter = x -> x.length() <= 4;
```

The lambda expression returns the value of the boolean expression `x.length() <= 4`. The filter function will return `true` when the length of the string `x` is at most 4, and `false` when the length of `x` is greater than 4.

**Code Listing 18-3** (`FilterDemo.java`)

```
import java.util.ArrayList;
import java.util.function.Predicate;

/**
```

```
   This program demonstrates the Collection removeIf.
*/

public class FilterDemo
{
    public static void main(String [] args)
    {
        // Array of names.
        String [] names = {"Anna", "Bob", "Carlos", "Debby"};

        // Create list and add names.
        ArrayList<String> nameList = new ArrayList<>();
        for (String name : names)
        {
            nameList.add(name);
        }
        // Use forEach with lambda expression to print.
        Predicate<String> filter = x -> x.length() <= 4;

        // Remove strings with length at most 4 from nameList
        nameList.removeIf(filter);

        // Print the array list to show remaining strings.
        System.out.println(nameList);
    }
}
```

**Program Output**

```
[Carlos, Debby]
```

**Checkpoint**

18.1 The interfaces and classes in the Java Collections Framework are part of what package?

18.2 What are the three general types of collections?

18.3 Describe the differences between a list and a set.

18.4 A map's elements are comprised of how many parts? What are they?

18.5 The `List` and `Set` interfaces extend what other interface?

18.6 Name two methods in the `Iterable` interface.

18.7 What is an iterator?

## 18.2 Lists

**CONCEPT: There are two different ways to implement the `List` interface. The `ArrayList` class uses an implementation based on contiguous memory allocation, while the `LinkedList` implementation is based on linked allocation.**

A list is a collection that stores its elements as a sequence. Each element in a sequence is assigned a nonnegative integer called its index. The first element in the sequence is assigned an index of 0, the next an index of 1, and so on.

## The `List` Interface

The `List` interface specifies the operations that a list must support. It extends the `Collection` interface, so it inherits all of the methods specified by `Collection`. It also adds additional methods, shown in Table 18-3, specifically for working with lists. In a nutshell, the `List` interface specifies methods for the following:

- Adding an element at a specific index
- Retrieving an element from a specific index
- Changing the value of an element at a specific index
- Removing an element that is at a specific index
- Searching for an element in the list and returning its index
- Retrieving a segment of the list as a sublist

**Table 18-3** Methods of the `List<E>` interface

| | |
|---|---|
| `void add(int index, E element)` | Adds `element` to this list at the position specified by `index`. |
| `boolean addAll(int index, Collection<? extends E> c)` | The parameter `c` is a `Collection` object. This method adds all of the elements of `c` to this list, starting at the position specified by `index`. Any elements that are currently stored in this list, beginning at the position specified by `index`, are shifted into positions of higher index. The method returns `true` if the elements of c were added to this list. It returns `false` otherwise. |
| `E get(int index)` | Returns a reference to the element at the position specified by `index`. |
| `int indexOf(Object o)` | Returns the position of the first occurrence of the object `o` in the list. If `o` does not exist in the list, the method returns − 1. |
| `int lastIndexOf(Object o)` | Returns the position of the last occurrence of the object `o` in the list. If `o` does not exist in the list, the method returns − 1. |
| `E remove(int index)` | This method removes the element at the position specified by `index`, and returns a reference to the element that was removed. |
| `E set(int index, E element)` | This method stores the object referenced by `element` at the position specified by `index`. The object currently stored at the specified position is replaced. |
| `void sort(Comparator<? super E> c)` | Sorts this list according to the order induced by the specified comparator. |
| `List<E> subList(int fromIndex, int toIndex)` | This method returns a `List` object containing the elements of this `List`, starting with the element at `fromIndex` and ending with the element at `toIndex`. |

Notice that many of the methods in Table 18-3 take an integer parameter for an index. The value of the index must be within the proper bounds, or the method will throw an `IndexOutBoundsException`. Elements already in the list have nonnegative indices that are less than the length of the list, so `get()`, `remove()`, and `set()` will throw this exception if the index passed to them is negative or greater or equal to the length of the list. The `add()` and `addAll()` methods throw the exception if the index is less than 0, or greater than the length of the list.

## The `ArrayList` and `LinkedList` Classes

The purpose of an interface is to allow different classes to provide different implementations for the methods defined in the interface. JCF provides two implementations of the `List` interface. One of these is the `ArrayList` class, which you encountered in Chapter 7. An array list internally uses a backing array to keep track of references to its elements. The `ArrayList` class overcomes the fixed-size limitation of arrays by automatically allocating a bigger array whenever its backing array is about to overflow, and copying all its elements to the bigger array. The `ArrayList` implementation is said to use *contiguous memory allocation* because all the elements in the backing array occupy successive locations in memory.

Another implementation of the `List` interface is the `LinkedList` class. This class uses *linked allocation*: it keeps track of a reference to the first element in the list, and each element contains a reference to its successor. These successor references are called *links*. Thus, the element at index 0 contains a link to the element at index 1, which contains a link to the element at index 2, and so on. The last element in the list has no successor, so its successor reference is `null`. A list that uses linked allocation is called a *linked list*. Linked lists are studied further in Chapter 19.

Because of the nature of linked allocation, it takes longer to access an element near the end of the list than one near the beginning. This is because to get to an element at position $k$, you must start at zero and follow the successor links through all elements of lower index. In contrast, `ArrayList` can jump directly to the desired element by using $k$ to index the backing array. Because of this, `ArrayList` is said to provide *random access* to its elements.

The `ArrayList` implementation performs best when most of the elements added to the list are added at the end (the position with the highest index), and most of the elements being removed are removed from the end. This is because inserting an element in the middle of the backing array requires space to be created for the new element by moving all elements with index equal or greater to the insertion point up by one. Similarly, deleting an element in the middle leaves a gap in the backing array, and requires all elements after the deletion point to move down by one to close the gap.

The `LinkedList` implementation performs best when many elements need to be added or removed at a position in the middle of the list. Such an operation can be carried out by reassigning a few successor links, unlike contiguous allocation where lots of elements may need to be shifted.

## Creating and Initializing Lists

To create a list using the `ArrayList<E>` implementation, you need to use one of the following constructors:

```
ArrayList()
ArrayList(int initialCapacity)
ArrayList(Collection<? extends E> c)
```

The first constructor creates an empty list with the default initial capacity of 10, while the second constructor creates a list with the specified initial capacity. The *capacity* of an array list is the maximum number of elements it can hold before its backing array overflows. The capacity should not be confused with the *size* of the list, which is the number of elements currently stored in the list. The last constructor creates a list containing the same elements as the specified collection, in the order returned by that collection's iterator.

To create a list using the `LinkedList<E>` implementation, you need to use one of its two constructors:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

These constructors allow you to create an empty list, or a list whose elements come from a specified collection.

If you need to initialize a list to a long list of values, you can save yourself some typing by first initializing an array and then copying the array elements to a list using a loop:

```
// Initialize array of names.
String [] namesArray = {"Anna", "Bob", "Carlos", "Debby"};

// Create empty list.
LinkedList<String> list1 = new LinkedList<>();

// Populate the list using a loop.
for (String s : namesArray)
{
    list1.add(s);
}
System.out.println(list1);
```

Here we have created a linked list, added some elements to it, and printed the list. All implementations of collections in Java provide a `toString()` method, so collections can be printed without resorting to a loop to print the individual elements of the collection.

We can simplify this code further by using the constructor that takes an existing collection to initialize a list. The `Arrays` class has a method

```
static <T> List<T> asList(T ... a)
```

that takes an array of a reference type `T` and returns a fixed-size list backed by that array. So you can create a list from an array like this:

```
List<String> aList = Arrays.asList(namesArray);
```

You cannot add elements or remove elements from a fixed-size list, but you can use it as a parameter to a `LinkedList` or `ArrayList` constructor. Thus, you can initialize a list with names from an array, and a new name to the list, and print the list like this:

```
String [] namesArray = {"Anna", "Bob", "Carlos", "Debby"};
List<String> fixedList = Arrays.asList(namesArray);
LinkedList list2 = new LinkedList<>(fixedList);
list2.add("Ephraim");
System.out.println(list2);
```

This code will print the line:

```
[Anna, Bob, Carlos, Debby, Ephraim]
```

Because the `asList` method takes a var-args parameter, you can shorten the code to create and initialize the list further:

```
LinkedList list2 =
     new LinkedList<>(Arrays.asList("Anna", "Bob", "Carlos", "Debby"));
```

## List Iterators

For lists, the iterator inherited from `Collection` returns list elements in order of increasing index, and its `next()` method always moves "forward" through the list. The `List<E>` interface defines a method:

```
ListIterator<E> listIterator()
```

When called on a list, this method returns a special *list iterator* that is able to move both forward and backward through a list. In addition to a `next()` method, a list iterator has a `previous()` method that returns an element while moving backward. A list iterator has a cursor position that always lies between the element that would be returned by a call to `previous()` and the element that would be returned a call to `next()`. Initially, the cursor is just before the element with index 0. When `next()` is called, the cursor returns the element at 0 and moves forward past the returned element to the position between elements 0 and 1. At this point, a call to `previous()` will return the element 0 and move the cursor back to the beginning, but a call to `next()` will return element 1 and move the cursor forward to the position between elements 1 and 2. If the cursor is already past the last element, a call to `next()` will throw `NoSuchElementException`. Likewise, a call to `previous()` will throw the same exception if the cursor position is before the element 0.

The `remove()` method in a list iterator removes from the list the element that was returned by the last call to `next()` or `previous()`. The `remove()` method can only be called once after a call to either of those two methods. A list iterator also has an `add()` method that adds a new element to the list at the current cursor position. The new element goes just after the element that would be returned by a call to `previous()`, and just before the element that would be returned by a call to `next()`.

The methods of a list iterator are defined in the `ListIterator` interface. This interface extends `Iterator` and adds the methods shown in Table 18-4.

**Table 18-4** Methods of the `ListIterator<E>` interface

| | |
|---|---|
| `void add(E e)` | Adds the specified element into the list at the current cursor position. |
| `E previous()` | Returns the element just before the cursor position and moves the cursor position backward. |
| `boolean hasPrevious()` | Returns `true` if there is an element before the current cursor position, `false` otherwise. |
| `int nextIndex()` | Returns the index of the element that would be returned by a subsequent call to `next()`. |
| `int previousIndex()` | Returns the index of the element that would be returned by a subsequent call to `previous()`. |
| `void set(E e)` | Replaces the last element returned by `next()` or `previous()`. |

The program in Code Listing 18-4 demonstrates how to use a list iterator to traverse a list and add a new element.

**Code Listing 18-4** `(ListIteratorDemo.java)`

```java
import java.util.*;

/**
  Demonstrates the list iterator
*/

public class ListIteratorDemo
{
    public static void main(String [] args)
    {
        // Create an array list to hold strings.
        String []names = {"Chris", "David", "Katherine", "Kenny"};
        List<String> nameList = new ArrayList<>(Arrays.asList(names));

        // Display the names in the list.
        System.out.println("Here are the original names.");
        System.out.println(nameList);

        // Get a list iterator.
        ListIterator<String> it = nameList.listIterator();

        // Add "Darlene" to the list right after "Chris".
        while (it.hasNext())
        {
            String str = it.next();
            // If the last name retrieved was "Chris"
            // then insert "Darlene".
            if (str.equalsIgnoreCase("Chris"))
            {
                it.add("Darlene");
                // We are done
                break;
            }
        }
        //Display the names in the list again.
        System.out.println("\nHere are the new names now.");
        System.out.println(nameList);
    }
}
```

**Program Output**

```
Here are the original names.
[Chris, David, Katherine, Kenny]

Here are the new names now.
[Chris, Darlene, David, Katherine, Kenny]
```

The program in Code Listing 18-5 demonstrates using the `set()` method to replace an element, and the `remove()` method to remove an element. In this program, the first name in the list is replaced with `"Alfonso"` and the last name is removed.

**Code Listing 18-5** (`ListIteratorDemo2.java`)

```
import java.util.*;

/**
   This program demonstrates setting and removing list
   elements with a ListIterator.
*/

public class ListIteratorDemo2
{
    public static void main(String[] args)
    {
        // Create an ArrayList to hold String objects.
        List<String> nameList = new ArrayList<>();

        // Add some names to the ArrayList.
        nameList.add("Chris");
        nameList.add("David");
        nameList.add("Katherine");
        nameList.add("Kenny");

        // Display the names in the list
        System.out.println("Here are the original names.");
        System.out.println(nameList);

        // Get a list iterator.
        ListIterator<String> it = nameList.listIterator();

        // Replace the first element with "Alfonso".
        if (it.hasNext())
        {
            // Get the first element.
            it.next();

            // Replace it.
            it.set("Alfonso");
        }
        // Move the iterator to the end of the list.
        while (it.hasNext())
        {
            it.next();
        }

        // Remove the last element that was retrieved.
        it.remove();

        // Display the names in the list again. The list
        // should read Alfonso, David, and Katherine.
        System.out.println("\nHere are the names now.");
        System.out.println(nameList);
    }
}
```

**Program Output**

```
Here are the original names.
[Chris, David, Katherine, Kenny]

Here are the names now.
[Alfonso, David, Katherine]
```

## Methods of the `LinkedList` Class

Linked allocation makes it easy to move *forward* in a list; you can easily pass from one element to the next by following the successor link. The Java implementation of a linked list is actually a doubly linked list. In addition to each element having a link to its successor, every element has a link to its *predecessor*. The predecessor links make it easy to move *backward* in the list. Also, in addition to keeping a reference to the first element in the list, the Java implementation keeps a reference to the last element of the list. This makes it as easy to locate the last element as it is to locate the first.

To understand why a linked list should maintain predecessor as well as successor links, imagine an application that prints the element of a linked list of length 5 in reverse order. Suppose that the application accesses the elements by index, using the `get()` method of the `List` interface.

```
String [] namesArray = {"Anna", "Bob", "Carlos", "Debby", "Ephraim"};
LinkedList list2 = new LinkedList<>(Arrays.asList(namesArray));

for (int k = list2.size()-1; k >= 0; k--)
{
    System.out.printf("%s ", list2.get(k));
}
```

With each call to `get(k)`, the list would start at index 0 and traverse $k$ links to get to the node at index $k$. Altogether, this code would require the traversal of $4 + 3 + 2 + 1 = 10$ links. For a long list, the code would need to traverse a lot of links.

To facilitate the backward and forward traversals of a linked list, and to guarantee quick access to the first and last elements, the `LinkedList` class adds several methods not found in `List`.

**Table 18-5** Methods of the `LinkedList<E>` class

| | |
|---|---|
| `void addFirst(E e)` | Inserts the specified element at the beginning of this list. |
| `void addLast(E e)` | Appends the specified element to the end of this list. |
| `E getFirst()` | Returns the first element in this list. |
| `E getLast()` | Returns the last element in this list. |
| `E removeFirst()` | Removes and returns the first element in this list. |
| `E removeLast()` | Removes and returns the last element in this list. |
| `Iterator<E> descendingIterator()` | Returns an iterator that traverses the list in reverse order. |

**NOTE:** The methods of Table 18-5 for getting and removing the first and last elements will throw `NoSuchElementException` if called on an empty list.

## Using an Interface Variable to Reference a `Collection` Object

Take a closer look at the statement in line 13 of Code Listing 18-5:

```
List<String> nameList = new ArrayList<>();
```

Notice that `nameList` is a `List` variable, not an `ArrayList` variable. Because the `ArrayList` class implements the `List` interface, we can use a `List` variable to reference an `ArrayList` object. This is a common practice. By always using interface variables to reference collection objects, our code will be easier to maintain if it becomes necessary to use a different concrete class to create the object.

Either one of the `ArrayList` and `LinkedList class` can be used to create a list. Internally, however, there are differences in the way that these classes handle list operations. Suppose we write a program that at first uses an `ArrayList` to store a large number of `String` objects. We use the following statement to instantiate the `ArrayList`:

```
List<String> myList = new ArrayList<>();
```

Also, assume that in the program we pass the `ArrayList` object as an argument to numerous methods. Each of the methods uses a `List` variable as a parameter, such as the following:

```
public void displayList(List<String> somelist)
{
    method code here...
}
```

Once we put the program into use, we determine that the `ArrayList` is performing slowly because a large number of objects are constantly being inserted in the middle. After doing a little research, we determine that the `LinkedList` class performs faster under these circumstances, and we decide to use it instead of the `ArrayList` class. Because the `LinkedList` class also implements the `List` interface, and we have only used `List` variables to reference the `ArrayList`, making the change is simple. In the statement that instantiates the list object, we only have to change `ArrayList` to `LinkedList`, as shown here:

```
List<String> myList = new LinkedList<>();
```

Changes in other parts of the program are not necessary because we only used `List` variables to reference the list object.

## Checkpoint

18.8 What kind of memory allocation is used by the `ArrayList` class?

18.9 What kind of memory allocation is used by the `LinkedList` class?

18.10 Why is it a common practice to use an interface variable to reference a collection object?

18.11 `ListIterator` is a subinterface of what other JCF interface?

18.12 When it comes to moving through the elements of a collection, describe a difference between an `Iterator` and a `ListIterator`.

18.13 When you need a list iterator, how do you get one?

18.14 When stepping through the elements of a list with a list iterator's `next` method, how do you determine whether there are more elements in the list to visit?

18.15 Which method should you call to determine `previous()` without throwing an exception?

18.16 What element does the `remove` method (in the `Iterator` interface) remove?

18.17 What element does the `set` method (in the `ListIterator` interface) change?

18.18 What is the difference between the *size* and *capacity* of an `ArrayList`?

18.19 Which concrete list class stores its elements in a `linked list`?

18.20 Which concrete list class would be a good choice to use when your application will be storing a large amount of data in a list, and a lot of insertions and/or deletions in the middle of the list are likely to take place?

## 18.3 Sets

**CONCEPT: A set is a collection that lets you quickly find a value in a large amount of data. A set stores elements in an unordered sequence, and does not allow duplicates.**

Sometimes when data is stored in a list, you know the exact location of specific items. For example, suppose we have a list that holds the total sales for each month of the year. The list has twelve elements, with the indices 0 through 11. We know that January's sales amount is stored at index 0, February's sales amount is stored at index 1, and so forth. If we want to retrieve the sales amount for December, we know that we have to retrieve the element at index 11. In the list, the sales for month $n$ is stored at index $n - 1$.

Sometimes, we need to retrieve a particular element from a collection, but we do not know the location of that element. For example, suppose we have a collection of `Customer` objects, and we need to search for a particular object by the customer's name. `Customer` objects are added to and removed from the collection on a daily basis, and we have no way of knowing where a particular object is stored. If the objects are stored in a list, finding a particular customer means sequentially searching the list until we find the element we are looking for. This could be a time-consuming process if there are a large number of objects in the list.

Perhaps a set is the best type of collection to use in this situation—if fast retrieval is important. Sets are useful when you have a large collection of data, and you must retrieve an element from the collection by searching for its value. The elements in a set are stored in a data structure that is optimized for this type of retrieval.

It is helpful to compare sets to lists. A list will allow you to append an element at the end, add an element at the beginning, or insert an element at a specific position within the list. In contrast, a set element is not associated with a position within the set. Sets are also different from lists in another way: they do not allow duplicate elements. Each element in a set must be unique.

### The Set Interface

The `Set` interface extends the `Collection` interface, so it inherits all of the methods specified by `Collection`. (See Table 18-2 for a description of the methods specified by the `Collection` interface.) The `Set` interface does not add any additional methods of its own, but it does place restrictions on many of the methods inherited from `Collection`. These restrictions enforce the requirement that a set cannot contain duplicate elements.

### The HashSet Class

The first concrete set class that we will study is the `HashSet` class, which implements the `Set` interface. The `HashSet` class uses hash codes to store its elements in a way that makes searching faster. To understand how a `HashSet` works, first you must understand the concept of a hash code.

A *hash code* is an integer value that is characteristic of an object. You use a *hashing algorithm* to calculate a hash code for an object. There are many different hashing algorithms, ranging from the simple to the complex. Although we will not conduct an in-depth study of hashing algorithms, later in this section we will look at some simple techniques for calculating hash codes.

Hashing algorithms typically use the data that is stored in an object to calculate the object's hash code. For example, if we were to calculate the hash code for a string, we would somehow use the characters making up the string in the calculation. Perhaps we would use each character's Unicode value in a mathematical formula that would give us an integer value.

Ideally, our hashing algorithm would produce different hash codes for different strings. For example, the hash code for `"Fred"` would be different from the hash code for `"Wilma"`.

In addition to producing different hash codes for different objects, an ideal hashing algorithm should produce the same hash code for objects that are equal in value. For example, suppose we have the following two `String` objects, `str1` and `str2`:

```
String str1 = new String("Fred");
String str2 = new String("Fred");
```

Although `str1` and `str2` are different objects, they contain equal data and a hashing algorithm should produce the same hash code for both.

In Java, every object has a `hashCode` method that is inherited from the `Object` class. This method returns a hash code for the object. The `hashCode` method that is inherited from the `Object` class, however, is rarely adequate because it returns a value that is based on the object's memory address. The value that is returned from the `Object` class's `hashCode` method is unique for every object, even objects that contain the same data. For this reason, the `Object` class's `hashCode` method must usually be overridden so that it returns the same hash code for equivalent objects.

Hash codes are at the heart of how the `HashSet` class works. When an object is added to a `HashSet`, the `HashSet` calls the object's `hashCode` method to get its hash code. Then it checks to see if an object with the same hash code has already been stored in the set. (Remember, a set does not allow duplicates.) If it finds that an object with the same hash code is already in the set, it performs one more test: it calls the object's `equals` method to see if it reports that the two objects are equal. If they are, the `HashSet` does not store the object in the set.

You might be wondering why a `HashSet` does not simply refuse to store an object as soon as it finds another object with the same hash code already in the set. Earlier, we said that an ideal hashing algorithm would produce different hash codes for different values. That is true, but hashing algorithms are not usually ideal. Occasionally, a hashing algorithm will produce the same hash code for two objects that are not equal. This is called a *collision*.

Because collisions can occur, the `HashSet` performs two tests, one using the `hashCode` method and the other using the `equals` method, to determine whether an object's value is already in the set.

Because collisions sometimes occur with hashing algorithms, it is possible that when a new object is added to a `HashSet` it will have the same hash code as an object already in the set, but it will not be equal to that object (the two objects will contain different data). In this case, the `HashSet` must allow the new object to be added to the set.

One popular way of thinking about hash codes is that they identify "buckets" in the `HashSet`. Each individual hash code has its own bucket, and all objects in the set that have that hash code are stored in that hash code's bucket. So, some buckets will have only one object stored in them, while other buckets will contain multiple objects. In pseudocode, the following is the procedure followed by the `HashSet` class to add a new object:

```
Call the new object's hashCode method to get the object's hash code.
If that hash code's bucket is empty, Then
    Store the object in the bucket for that hash code.
Else
    Use the object's equals method to compare it to each object in the bucket.
    If it is not equal to any other object in the bucket, Then
        Store the object in the bucket.
    Else
        Do not store the new object.
    End If
End If
```

> **TIP:** Because the `hashCode` and `equals` methods are used together like this, any time you override the `Object` class's `equals` method you should also override the `hashCode` method.

The `HashSet` class has four constructors, which are summarized in Table 18-6. Note that when a `HashSet` is constructed, it has an initial capacity and a load factor. The *initial capacity* is the number of "buckets" that the `HashSet` initially has. The *load factor* is the percentage of the buckets that have to be occupied before the capacity is increased. For example, with a load factor of 0.75, the capacity will be increased when 75 percent of the buckets have a value stored in them.

**Table 18-6** HashSet constructors

| | |
|---|---|
| `HashSet()` | This constructor creates an empty `HashSet` object with an initial capacity of 16 and a load factor of 0.75. |
| `HashSet(Collection<? extends E> c)` | This constructor creates a `HashSet` object and stores the elements of the collection `c` in it. The initial capacity will be at least enough to hold the elements of `c`, and the load factor will be 0.75. Throws a `NullPointerException` if `c` is a null reference. |
| `HashSet(int initialCapacity, float loadFactor)` | This constructor creates an empty `HashSet` object with the specified initial capacity and load factor. |
| `HashSet(int initialCapacity)` | This constructor creates an empty `HashSet` object with the specified initial capacity and a load factor of 0.75. |

Now, let's look at a simple example of how we can store `String` objects in a `HashSet`. The program in Code Listing 18-6 creates a `HashSet` and adds a group of `String` objects to it. Then it attempts to add a duplicate `String` object to the set.

**Code Listing 18-6** `(HashSetDemo1.java)`

```java
import java.util.*;

/**
  This program demonstrates how to add elements to a HashSet.
  It also shows that duplicate elements are not allowed.
*/
public class HashSetDemo1
{
    public static void main(String[] args)
    {
        // Create a HashSet to hold String objects.
        Set<String> fruitSet = new HashSet<>();

        // Add some strings to the set.
        fruitSet.add("Apple");
        fruitSet.add("Banana");
        fruitSet.add("Pear");
        fruitSet.add("Strawberry");

        // Display the elements in the set.
        System.out.println("Here are the elements.");
        System.out.println(fruitSet);

        // Try to add a duplicate element.
        System.out.println("\nTrying to add Banana to"
                + "the set again . . .");
        if (!fruitSet.add("Banana"))
        {
            System.out.println("Banana was not added again.");
        }

        // Display the elements in the set.
        System.out.println("\nHere are the elements once more.");
        System.out.println(fruitSet);
    }
}
```

**Program Output**

```
Here are the elements.
[Apple, Pear, Strawberry, Banana]

Trying to add Banana to the set again...
Banana was not added again.

Here are the elements once more.
[Apple, Pear, Strawberry, Banana]
```

Take a closer look at lines 27–30:

```
if (!fruitSet.add("Banana"))
{
    System.out.println("Banana was not added again.");
}
```

The `HashSet` class's `add` method does not throw an exception if it is called with a duplicate item. It simply does not add the item. The method does return a boolean value, however, indicating whether the item was added. The statement shown here calls the `add` method, passing `"Banana"` as an argument. Because `"Banana"` is already in the set, the method returns false indicating that it was not added again.

The program in Code Listing 18-7 shows another demonstration. It uses the second constructor shown in Table 18-6 to create a `HashSet` from a previously created collection of names, and then gets an iterator to step through the set and print its elements. The program also calls the `HashSet` class's `contains` method to determine whether an object has been stored in the set.

**Code Listing 18-7**

```
import java.util.*;

/**
  This program creates a HashSet with names in it,
  gets an iterator for the set, and searches the
  set for names.
*/

public class HashSetDemo2
{
    public static void main(String[] args)
    {
        // Create a HashSet to hold names.
        List<String> nameList =
                Arrays.asList("Chris", "David", "Katherine", "Kenny");
        Set<String> nameSet = new HashSet<>(nameList);

        // Get an iterator for the set.
        Iterator<String> it = nameSet.iterator();

        // Display the elements in the set.
        System.out.println("Here are the names in the set.");
        while (it.hasNext())
        {
            System.out.println(it.next());
        }
        System.out.println();

        // Search for "Katherine". We should find this
        // name in the set.
        if (nameSet.contains("Katherine"))
        {
            System.out.println("Katherine is in the set.");
        } else
        {
            System.out.println("Katherine is NOT in the set.");
        }

        // Search for "Bethany". We should not find
        // this name in the set.
        if (nameSet.contains("Bethany"))
        {
            System.out.println("Bethany is in the set.");
        } else
        {
            System.out.println("Bethany is NOT in the set.");
        }
    }
}
```

**Program Output**

```
Here are the names in the set.
Chris
Katherine
David
Kenny

Katherine is in the set.
Bethany is NOT in the set.
```

## Implementing Your Own hashCode Method

When you design a class, you will need to override the `hashCode` and `equals` methods if you want to store objects of the class in a `HashSet`. Here are three guidelines that you should follow when writing your own `hashCode` and `equals` methods:

1. If you call an object's `hashCode` method more than once during the execution of an application, the method should consistently return the same value. The only exception is when something in the object has changed, which would affect the way the `equals` method makes comparisons.
2. If the `equals` method reports that two objects are equal, then both objects' `hashCode` methods should return the same value.
3. If the `equals` method reports that two objects are *not* equal, it is permissible for both objects to have the same hash code. You should be aware, however, that having too many unequal objects with the same hash code will degrade the performance of an algorithm that uses the hash codes for searching.

The `String` class uses the following formula to calculate a hash code for a string:

$$hashCode = S_0 * 31^{(n-1)} + S_1 * 31^{(n-2)} + \ldots + S_{n-1}$$

In the formula, $n$ is the number of characters in the string, $S_0$ is the Unicode value of the character at position 0, $S_1$ is the Unicode value of the character at position 1, and $S_{n-1}$ is the Unicode value of the character at position $n - 1$.

The algorithm used by the `String` class is moderately complex. Other classes use a simpler approach. For example, the `Character` class simply uses the character's Unicode value as its hash code. The `Integer` class uses the actual integer value of the object as its hash code.

Code Listing 18-8 shows a `Car` class that we will use to hold a VIN (vehicle identification number) and description of a car. These fields are stored as the `String` objects `vin` and `description`. In this class, we have overridden the `hashCode` method to simply return the hash code of `vin`. This should be adequate because each car has its own unique VIN. We have also overridden the `equals` method to return `true` if both objects have the same VIN.

**Code Listing 18-8** (`Car.java`)

```
/**
  The Car class stores a VIN (Vehicle Identification Number)
  and a description for a car.
*/
public class Car
{
     private String vin; // Vehicle Identification Number
     private String description; // Car description

     /**
       Constructor

       @param v The VIN for the car.
       @param desc The description of the car.
     */
    public Car(String v, String desc)
    {
         vin = v;
         description = desc;
    }

    /**
       getVin method

       @return The car's VIN.
    */
    public String getVin()
    {
         return vin;
```

```
   }

   /**
     getDescription method

     @return The car's description.
   */
   public String getDescription()
   {
      return description;
   }

   /**
     toString method

     @return A string containing the VIN and description.
   */
   public String toString()
   {
      return "VIN:" + vin
              + "\tDescription:"
              + description;
   }

   /**
     hashCode method

     @return A hash code for this car.
   */
   public int hashCode()
   {
      return vin.hashCode();
   }

   /**
     equals method

     @param obj Another object to compare this object to.
     @return true if the two objects are equal, false otherwise.
   */
   public boolean equals(Object obj)
   {
      // Make sure the other object is a Car.
      if (obj == null || !(obj instanceof Car) )
         return false;
      else
      {
         // Both are cars: Get a reference to other
```

```java
            // object as a car and check if their VIN
            // numbers are the same.
            Car tempCar = (Car) obj;
            return vin.equalsIgnoreCase(tempCar.vin);
        }
    }
}
```

The program in Code Listing 18-9 tests the class by adding several `Car` objects to a `HashSet`. Then the `HashSet` is searched for two `Car` objects: one that is in the set, and one that is not.

**Code Listing 18-9** (`CarHashSet.java`)

```java
import java.util.*;

/**
  This program stores Car objects in a HashSet and then
  searches for various objects.
*/
public class CarHashSet
{
    public static void main(String[] args)
    {
        // Create a HashSet to store Car objects.
        Set<Car> carSet = new HashSet<>();

        // Add some Car objects to the HashSet.
        carSet.add(new Car("227H54", "1997 Volkswagen"));
        carSet.add(new Car("448A69", "1965 Mustang"));
        carSet.add(new Car("453B55", "2007 Porsche"));
        carSet.add(new Car("177R60", "1980 BMW"));

        // Display the elements in the HashSet.
        System.out.println("Here are the cars in the set:");
        for (Car c : carSet)
        {
            System.out.println(c);
        }
        System.out.println();

        // Search for a specific car. This one is in the set.
        Car mustang = new Car("448A69", "1965 Mustang");
        System.out.println("Searching for" + mustang);
        if (carSet.contains(mustang))
        {
            System.out.println("The Mustang is in the set.");
        } else
        {
            System.out.println("The Mustang is NOT in the set.");
        }

        // Search for another car. This one is not in the set.
        Car plymouth = new Car("911C87", "2000 Plymouth");
        System.out.println("Searching for" + plymouth);
        if (carSet.contains(plymouth))
        {
            System.out.println("The Plymouth is in the set.");
        } else
        {
            System.out.println("The Plymouth is NOT in the set.");
        }
    }
}
```

**Program Output**

```
Here are the cars in the set:
VIN: 177R60  Description: 1980 BMW
VIN: 227H54  Description: 1997 Volkswagen
VIN: 448A69  Description: 1965 Mustang
VIN: 453B55  Description: 2007 Porsche

Searching for VIN: 448A69 Description: 1965 Mustang
The Mustang is in the set.
Searching for VIN: 911C87 Description: 2000 Plymouth
The Plymouth is NOT in the set.
```

## The `LinkedHashSet` Class

You might have noticed that in the program output of Code Listings 18-6, 18-7, and 18-9, the order in which the `HashSet` elements are displayed does not match the order in which they were added to the set. Sets have no concept of internal position of their elements, and they do not keep track of the order of insertion. When you iterate through a `HashSet`, you should not expect the iterator to return elements in the order of their insertion.

If you need to access set elements in the order that they were inserted, you can use the `LinkedHashSet` class instead of `HashSet`. The `LinkedHashSet` class, which extends `HashSet`, keeps an internal linked list that references the elements of the set in the order that they were inserted. This gives you the ability of iterating through the set in that order.

The `LinkedHashSet` class's constructors and methods are identical to those of the `HashSet` method. The program in Code Listing 18-10 creates two sets: a `HashSet` and a `LinkedHashSet`. The same `String` objects are stored in each set.

**Code Listing 18-10** (`LinkedHashSetDemo.java`)

```java
import java.util.*;

/**
  This program demonstrates how a LinkedHashSet
  preserves the order in which elements are added.
*/
public class LinkedHashSetDemo
{
    public static void main(String[] args)
    {
        // List of fruits to put in hash sets.
        List<String> fruitList =
              Arrays.asList("Apple", "Strawberry", "Pear", "Banana");

        // Create the HashSet.
        Set<String> set1 = new HashSet<>(fruitList);

        // Now create the LinkedHashSet.
        Set<String> set2 = new LinkedHashSet<>(fruitList);

        // Display the elements in the HashSet.
        System.out.println("Here are the elements in"
                + "the HashSet:");
        System.out.println(set1);

        // Now display the elements in the LinkedHashSet.
        System.out.println("\nAnd here are the elements"
                + "in the LinkedHashSet:");
         System.out.println(set2);
    }
}
```

**Program Output**

```
Here are the elements in the HashSet:
[Apple, Pear, Strawberry, Banana]

And here are the elements in the LinkedHashSet:
[Apple, Strawberry, Pear, Banana]
```

## The `SortedSet` Interface and the `TreeSet` Class

The `Set` interface has a subinterface called `SortedSet` whose methods are shown in Table 18-7.

**Table 18-7** `SortedSet<E>` methods

| | |
|---|---|
| `Comparator <? super E> comparator()` | This method returns a reference to the comparator used with this `SortedSet`. We discuss comparators later in this section. |
| `E first()` | Returns the first element in this set. This will be the element with the lowest value. |
| `SortedSet<E> headSet(E toElement)` | Returns a `SortedSet` that serves as a "view" into this `SortedSet`. The elements in the set that are returned are all less in value than the element specified by `toElement`. The elements in the set that are returned are merely references to elements in this set, so any changes made to the elements in the returned set are reflected in this set. |
| `E last()` | Returns the last element in this set. This will be the element with the highest value. |
| `SortedSet<E> subSet(E fromElement, E toElement)` | Returns a `SortedSet` that serves as a "view" into this `SortedSet`. The elements in the set that are returned are greater than or equal to the element specified by `fromElement`, and less than the element specified by `toElement`. The elements in the set that are returned are merely references to elements in this set, so any changes made to the elements in the returned set are reflected in this set. |
| `SortedSet<E> tailSet(E fromElement)` | Returns a `SortedSet` that serves as a "view" into this `SortedSet`. The elements in the set that are returned are all greater in value than the element specified by `fromElement`. The elements in the set that are returned are merely references to elements in this set, so any changes made to the elements in the returned set are reflected in this set. |

A `SortedSet` is a set in which the elements are sorted according to their natural order. A class of objects has a *natural order* if it is possible to determine for any two objects of the class whether one object is equal to, or "less than," or "greater than" the other. In this context, *less than* and *greater than* refers to which element should precede the other on the natural ordering of that class. For example, the natural order of numbers is to order them in increasing order. For strings, the natural order is alphabetic order.

A class imposes a natural order on its objects by implementing the `Comparable` interface. We looked at this interface in Chapter 16 when we were studying generics. Recall that the `Comparable` interface specifies a method named `compareTo`, which is used to compare the calling object with another object that is passed as an argument to the method. This is how the `compareTo` method works:

- If the calling object is less than the object passed as an argument, the method returns a negative number.
- If the calling object is equal to the object passed as an argument, the method returns 0.
- If the calling object is greater than the object passed as an argument, the method returns a positive number.

There is a JCF class called `TreeSet` that implements the `SortedSet` interface. If you create a `TreeSet` using that class's no-arg constructor, the tree set will sort objects added to it by their natural order. The program in Code Listing 18-11 demonstrates this by creating a `TreeSet` and using it to sort strings. This works because the `String` class implements the `Comparable` interface.

**Code Listing 18-11** (`TreeSetDemo1.java`)

```java
import java.util.*;

/**
  This program demonstrates how a TreeSet sorts
  its elements in ascending order.
*/
public class TreeSetDemo1
{
    public static void main(String[] args)
    {
        // Create a TreeSet and store some values in it.
        SortedSet<String> mySet = new TreeSet<>();
        mySet.add("Pear");
        mySet.add("Apple");
        mySet.add("Strawberry");
        mySet.add("Banana");

        // Display the elements in the TreeSet.
        System.out.println("Here are the TreeSet elements"
                + "in ascending order:");
        System.out.println(mySet);

        // Add a new element to the TreeSet.
        System.out.println("\nAdding Blueberry to the set.");
        mySet.add("Blueberry");

        // Display the elements again.
        System.out.println("\nHere are the TreeSet elements"
                + "again:");
        System.out.println(mySet);
    }
}
```

**Program Output**

```
Here are the TreeSet elements in ascending order:
[Apple, Banana, Pear, Strawberry]

Adding Blueberry to the set.

Here are the TreeSet elements again:
[Apple, Banana, Blueberry, Pear, Strawberry]
```

## Using a Comparator to Compare Objects

If the objects that you wish to store in a `TreeSet` do not implement the `Comparable` interface, you can still store them in the `TreeSet` by creating a comparator. A *comparator* is an object that can compare two other objects and determine whether one is greater than, less than, or equal to the other. A comparator implements the `Comparator` interface, which is in the `java.util` package. The `Comparator` interface has two abstract methods:

```
public interface Comparator<T>
{
    public int compare(T obj1, T obj2);
    public boolean equals(Object obj);
}
```

**NOTE:** Beginning with Java 8, the `Comparator` interface contains other methods besides `compare` and `equals`.

Table 18-8 describes the two abstract methods specified by the `Comparator` interface.

**Table 18-8** Abstract methods of the `Comparator<T>` interface

| | |
|---|---|
| `public int compare(T obj1, T obj2)` | This method compares `obj1` with `obj2` and returns 0 if the two objects are equal, a negative value if `obj1` is less than `obj2`, and a positive value if `obj1` is greater than `obj2`. |
| `public boolean equals(Object obj)` | This method determines whether this comparator is equal to the object `obj`. If the two are equal, it returns `true`. |

Code Listing 18-12 shows an example of a `Comparator` class that can compare `Car` objects. The compare method accepts references to two `Car` objects. It gets each car's VIN, which is a `String`, and then uses the `String` class's `compareToIgnoreCase` method to compare the two VINs.

**Code Listing 18-12** (`CarComparator.java`)

```
import java.util.Comparator;

public class CarComparator
        implements Comparator<Car>
{
    public int compare(Car car1, Car car2)
    {
        // Get the two cars' VINs.
        String vin1 = car1.getVin();
        String vin2 = car2.getVin();

        // Compare the VINs and return the
        // result of the comparison.
        return vin1.compareToIgnoreCase(vin2);
    }
}
```

**NOTE:** We did not provide an `equals` method in the `CarComparator` class, even though it is specified in the `Comparator` interface. The `Object` class provides an `equals` method, so we will not get an error when we compile this class. If we wanted to be able to compare objects of this class with other comparators, we would have provided an `equals` method.

The program in Code Listing 18-13 demonstrates how to create a `TreeSet` that uses a `CarComparator` to compare its elements. If you pass a comparator object as an argument to the `TreeSet` constructor, the `TreeSet` will use the comparator to perform comparisons.

**Code Listing 18-13** (`TreeSetDemo2.java`)

```
import java.util.*;

/**
  This program demonstrates how a TreeSet can use
  a Comparator to sort its elements.
*/
public class TreeSetDemo2
{
    public static void main(String[] args)
    {
        // Create a TreeSet and pass an instance of
        // CarComparator to it.
        SortedSet<Car> carSet
                = new TreeSet<>(new CarComparator());

        // Add some Car objects to the TreeSet.
        carSet.add(new Car("227H54", "1997 Volkswagen"));
        carSet.add(new Car("453B55", "2007 Porsche"));
        carSet.add(new Car("177R60", "1980 BMW"));
        carSet.add(new Car("448A69", "1965 Mustang"));

        // Display the elements in the TreeSet.
        System.out.println("Here are the cars sorted in"
                + "order of their VINs:");
        for (Car car : carSet)
        {
            System.out.println(car);
        }
    }
}
```

**Program Output**

```
Here are the cars sorted in order of their VINs:
VIN: 177R60  Description: 1980 BMW
VIN: 227H54  Description: 1997 Volkswagen
VIN: 448A69  Description: 1965 Mustang
VIN: 453B55  Description: 2007 Porsche
```

Table 18-9 summarizes some of the most frequently used `TreeSet` constructors.

**Table 18-9** `TreeSet<E>` constructors

| | |
|---|---|
| `TreeSet()` | This constructor creates an empty `TreeSet` object. Any objects added to the `TreeSet` must implement the `Comparable` interface. |
| `TreeSet(Collection<? extends E> c)` | This constructor creates a `TreeSet` object and stores the elements of the collection `c` in it. The elements of `c` must implement the `Comparable` interface. |
| `TreeSet(Comparator<? extends E> c)` | This constructor creates a `TreeSet` object and uses `c`, a comparator, to compare its elements. |

**Checkpoint**

18.21 What is a hash code?

18.22 What method do you call to get an object's hash code?

18.23 In hashing, what is a collision?

18.24 What does the `HashSet` class's `add` method do when you try to add a duplicate item?

18.25 Any time you override the `Object` class's `equals` method, what other method should you also override?

18.26 Which concrete set class should you use if you need to access its elements in the order that they were inserted?

18.27 Which concrete set class should you use if you need to retrieve its elements in their natural order?

18.28 What is a comparator? What interface must it implement?

## 18.4 Maps

**CONCEPT: Each element in a map has two parts: a key and a value. Each key is associated with a specific value, and can be used to locate that value.**

Each element in a *map* has two parts: a key and a value. Each key is associated with a specific value, and can be used to locate that value. For example, suppose we wish to store employee names in a map, and we want to search for specific employees by their ID numbers. Each time we store an employee's name in the map as a value, we also store that employee's ID number as a key. The map creates an association between the value and the key, and allows us to retrieve the name by searching for the ID number. The elements in a map are referred to as *mappings*.

There are a number of similarities between sets and maps. In fact, you can think of a map as storing a set of keys, with each key in the map's *key set* being associated with a value. Just as the JCF has a `Set` interface, there is a `Map` interface. Just as the `HashSet` and `LinkedHashSet` classes implement the `Set` interface; there are `HashMap` and `LinkedHashMap` classes that implement the `Map` interface. Just as `Set` has a `SortedSet` subinterface; `Map` has a `SortedMap` subinterface. Finally, there is a `TreeMap` class that implements the `SortedMap` interface, mirroring the way that `TreeSet` implements the `SortedSet` interface.

The map interface is generic, and is parameterized by two type variables `K` and `V`. These respectively stand for the classes representing the key and value objects. Table 18-10 lists some of the more commonly used methods specified by the `Map<K,V>` interface.

**Table 18-10** Methods specified by the Map<K,V> interface

| | |
|---|---|
| `void clear()` | Clears the map of all its elements. (This is an optional method. If the map does not support it, it throws an `UnsupportedOperationException`.) |
| `boolean containsKey(Object key)` | Returns `true` if this map contains a value that is associated with the specified key. |
| `boolean containsValue(Object value)` | Returns `true` if this map contains one or more keys mapped to the specified value. |
| `Set< Map.Entry<K,V> > entrySet()` | Returns a set that serves as a "view" into this map, containing all of the map's mappings. The elements in the set that is returned all implement the `Map.Entry` interface. (`Entry` is defined as an inner interface in the `Map` interface.) The elements in the set that is returned are merely references to elements in this set, so any changes made to the elements in the returned set are reflected in this map. |
| `void forEach(BiConsumer<? super K,? super V> action)` | Calls the action method on each key and value pair in this map. |
| `V get(Object key)` | Returns the value that is associated with the specified key. If the key does not map to a value, the method returns null. |
| `boolean isEmpty()` | Returns `true` if this map contains no mappings. |
| `Set<K> keySet()` | Returns a set that contains all of the keys in this map. The set that is returned serves as a "view" into this map. The elements in the set are merely references to the keys in this map, so any changes made to the elements in the returned set are reflected in this map. |

*(table continues next page)*

**Table 18-10** Methods specified by the Map<K, V> interface (continued)

| | |
|---|---|
| `V put(K key, V value)` | This method puts a mapping in the map. The specified key is mapped to the specified value. If an existing mapping for this key exists, it is replaced by this mapping. The method returns the previous value that was associated with this key, or `null` if there was no value associated with this key previously. |
| `V remove(Object key)` | If a mapping for the specified key exists, it is removed. A reference to the value that the key was previously mapped to is returned. |
| `int size()` | Returns the number of mappings in this map. |
| `Collection<V> values()` | Returns a collection that contains all of the values in this map. The collection that is returned serves as a "view" into this map. The elements in the collection are merely references to the values in this map, so any changes made to the elements in the returned collection are reflected in this map. |

## The HashMap Class

VideoNote The HashMap Class

The `HashMap` class is a concrete class that implements the `Map` interface. Like a `HashSet`, a `HashMap` uses hash codes to store and retrieve data. Specifically, the `HashMap` class stores keys by their hash codes. Table 18-11 summarizes the `HashMap` constructors. Some of these constructors will look familiar because they are similar to the `HashSet` constructors. Because the `HashMap` class uses hash codes to store keys, it provides constructors that allow you to specify initial capacity and load factor. (See the previous section on the `HashSet` class for a discussion of initial capacity and load factor.)

**Table 18-11** HashMap<K, V> constructors

| | |
|---|---|
| `HashMap()` | This constructor creates an empty `HashMap` object with an initial capacity of 16 and a load factor of 0.75. |
| `HashMap(Map<? Extends K, ? extends V> m)` | This constructor creates a `HashMap` object and stores the mappings of the map `m` in it. The initial capacity will be at least enough to hold the elements of `m`, and the load factor will be 0.75. Throws a `NullPointerException` if `m` is a `null` reference. |
| `HashMap(int initialCapacity, float loadFactor)` | This constructor creates an empty `HashMap` object with the specified initial capacity and load factor. |
| `HashMap(int initialCapacity)` | This constructor creates an empty `HashMap` object with the specified initial capacity and a load factor of 0.75. |

The program in Code Listing 18-14 demonstrates a simple `HashMap` for storing `Car` objects. This application uses VINs (vehicle identification numbers) as keys, and `Car` objects as values. As a result, the `HashMap` will allow us to search for a specific `Car` object by specifying its VIN.

**Code Listing 18-14** (`CarHashMap1.java`)

```
import java.util.*;

/**
   This program stores mappings in a HashMap and then
   searches for various objects.
*/

public class CarHashMap1
{
    public static void main(String[] args)
   {
       // Create a HashMap to store Car objects.
       Map<String, Car> carMap = new HashMap<>();

       // Create some Car objects.
       Car vw = new Car("227H54", "1997 Volkswagen");
       Car mustang = new Car("448A69", "1965 Mustang");
       Car porsche = new Car("453B55", "2007 Porsche");
       Car bmw = new Car("177R60", "1980 BMW");

       // Put some mappings into the HashMap. In each
       // mapping, the car's VIN is the key and the
       // Car object containing that VIN is the value.
       carMap.put(vw.getVin(), vw);
       carMap.put(mustang.getVin(), mustang);
       carMap.put(porsche.getVin(), porsche);
       carMap.put(bmw.getVin(), bmw);

       // Search for the Mustang by its VIN.
       System.out.println("\nSearching for the car with" +
                          "VIN" + mustang.getVin());
       Car foundCar = carMap.get(mustang.getVin());

       // If the car was found, display it.
       if (foundCar != null)
           System.out.println(foundCar);
       else
           System.out.println("The Mustang is NOT in the set.");

       // Search for another VIN. This one is not in the set.
       System.out.println("\nSearching for the car with" +
                          "VIN 911C87");
       foundCar = carMap.get("911C87");

       // If the car was found display it.
46          if (foundCar != null)
47             System.out.println(foundCar);
48          else
49             System.out.println("That car is NOT in the set.");
50       }
51    }
```

**Program Output**

```
Searching for the car with VIN 448A69
VIN: 448A69 Description: 1965 Mustang

Searching for the car with VIN 911C87
That car is NOT in the set.
```

The statement in line 13 creates the `HashMap`:

```
Map<String, Car> carMap = new HashMap<>();
```

Note that the `Map<K, V>` interface has been specialized to `Map<String, Car>` because `String` is the data type of the keys and `Car` is the data type of the values.

The statements in lines 16 through 19 create four `Car` objects: one for a VW, one for a Mustang, one for a Porsche, and one for a BMW. Then, the statements in lines 24 through 27 use the `put` method to create several mappings in the `HashMap`. For example, look at the statement in line 24:

```
carMap.put(vw.getVin(), vw);
```

The `put` method puts a mapping into the `HashMap`. The first argument is the key, and the second argument is the value to associate with the key. In this program, we are creating mappings between VINs and `Car` objects. In each mapping, the key is a VIN and the value is the `Car` object that goes with that VIN. In this statement, we are creating a mapping that associates the VW's VIN with the VW's `Car` object. The statements in lines 24 through 27 create similar mappings for the other `Car` objects.

Line 32 shows how we can search for a specific `Car` object:

```
Car foundCar = carMap.get(mustang.getVin());
```

This statement creates a `Car` reference variable, `foundCar`, and assigns to it the value returned from the `carMap.get` method. The `get` method accepts a key as its argument and it returns the value that the key is mapped to. In this statement, we are passing the Mustang's VIN as the argument. We are basically saying to the `HashMap` "Give me the `Car` object that is associated with this VIN." If the key is found in the `HashMap`, the method returns a reference to the value that is associated with that key. Because the Mustang's VIN is indeed found as a key in the `HashMap`, this statement returns a reference to the mustang object and assigns to `foundCar`.

In line 43 we aren't so lucky. Here we pass to the `get` method a VIN that does not exist in the `HashMap`:

```
foundCar = carMap.get("911C87");
```

As a result, the method will return `null`.

Unlike the `Set<E>` and `List<E>` interfaces, `Map<K, V>` does not extend the `Collection<E>` interface. And unlike the other collections, maps do not provide iterators. One way to iterate over the contents of a map is to convert its mappings to some other type of collection. For example, you can use the `keyset()` method to get a set that contains all of the keys in a map, and you can use the `values()` method to get a collection that contains all of the map's values. Then you can iterate over the resulting set or collection. Code Listing 18-15 shows an example.

**Code Listing 18-15** (CarHashMap2.java)

```
import java.util.*;

/**
   This program retrieves a set of keys and a
   collection of values from a HashMap.
*/

public class CarHashMap2
{
    public static void main(String[] args)
    {
        // Create a HashMap to store Car objects.
        Map<String, Car> carMap =  new HashMap<>();

       // Create some Car objects.
       Car vw = new Car("227H54", "1997 Volkswagen");
       Car mustang = new Car("448A69", "1965 Mustang");
       Car porsche = new Car("453B55", "2007 Porsche");
       Car bmw = new Car("177R60", "1980 BMW");

       // Put some mappings into the HashMap. In each
       // mapping, the car's VIN is the key and the
       // Car object containing that VIN is the value.
       carMap.put(vw.getVin(), vw);
       carMap.put(mustang.getVin(), mustang);
       carMap.put(porsche.getVin(), porsche);
       carMap.put(bmw.getVin(), bmw);

       // Get a set containing the keys in this map.
       Set<String> keys = carMap.keySet();

       // Iterate through the keys, printing each one.
       System.out.println("Here are the keys:");
       for (String k : keys)
           System.out.println(k);

       // Get a collection containing the values.
       Collection<Car> values = carMap.values();

       // Iterate through the values, printing each one.
       System.out.println("\nHere are the values:");
       for (Car c : values)
          System.out.println(c);
   }
}
```

**Program Output**

```
Here are the keys:
177R60
227H54
448A69
453B55

Here are the values:
VIN: 177R60   Description: 1980 BMW
VIN: 227H54   Description: 1997 Volkswagen
VIN: 448A69   Description: 1965 Mustang
VIN: 453B55   Description: 2007 Porsche
```

**NOTE:** The set and the collection that are returned from the `keyset` and `values` methods serve as "views" into the map. The elements in the set and collection are merely references to the entries in the map, so any changes made to the elements in the set or collection are reflected in the map, and vice versa.

In Code Listing 18-15, the keys and values of the map are retrieved separately. You can also retrieve a single set that contains all the key-value pairs stored in the map. This is done by the map's `entrySet` method.

The `entrySet` method returns a set of `Map.Entry` objects. `Entry` is an inner interface in the `Map` interface. Two of the methods specified by `Map.Entry` are `getKey` and `getValue`. The `getKey` method returns the key and `getValue` returns the value. The program in Code Listing 18-16 demonstrates the `entrySet` method and how to access the keys and values of the `Set` elements it returns.

**Code Listing 18-16** (`CarHashMap3.java`)

```
import java.util.*;

/**
   This program retrieves the mappings from a HashMap
   as a Set of Map.Entry objects.
*/

```

```
public class CarHashMap3
{
   public static void main(String[] args)
   {
      // Create a HashMap to store Car objects.
      Map<String, Car> carMap = new HashMap<>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69", "1965 Mustang");
      Car porsche = new Car("453B55", "2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");

      // Put some mappings into the HashMap. In each
      // mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);

      // Get a set containing the mappings in this map.
      Set<Map.Entry<String, Car>> cars = carMap.entrySet();

      // Iterate through the mappings, printing each one.
      System.out.println("Here are the mappings:");
      for (Map.Entry<String, Car> entry : cars)
      {
         System.out.println("Key =" + entry.getKey());
         System.out.println("Value =" + entry.getValue());
         System.out.println();
      }
   }
}
```

**Program Output**

```
Here are the mappings:
Key = 177R60
Value = VIN: 177R60   Description: 1980 BMW

Key = 227H54
Value = VIN: 227H54   Description: 1997 Volkswagen

Key = 448A69
Value = VIN: 448A69   Description: 1965 Mustang

Key = 453B55
Value = VIN: 453B55   Description: 2007 Porsche
```

**NOTE:** The set that is returned from the `entrySet()` method also serves as a "view" into the map. The elements in the set are merely references to the entries in the map, so any changes made to the elements in the set are reflected in the map, and vice versa.

Java 8 added to the `Map<K, V>` interface a default method

```
void forEach(BiConsumer<? super K,? super V> action)
```

that can be used to invoke an action on each key-value pair stored in a map. The action must implement the `BiConsumer<? super K,? super V>` interface. This means that the `action` function must have a type that is appropriately related to the type (***K***, ***V***) → `void`; its parameters are a key and a value, and its return type is `void`. As usual, we can use a lambda expression for the action. If we adopt this technique, Lines 29-39 of Code Listing 18–16 can be replaced with:

```
System.out.println("Here are the mappings:");
carMap.forEach( (key, value) ->
                {
                   System.out.println("Key = " + key);
                   System.out.println("Value = " + value);
                   System.out.println();
                });
```

## The `LinkedHashMap` Class

`Hashmap` objects, like sets, have no concept of order for their elements. You will notice, for example, that the order in which the mappings are displayed in Code Listing 18-16 does not match the order in which they were put into the `HashMap`.

The `LinkedHashMap` class, which extends `HashMap`, allows you to access its mappings in one of two orders:

- Insertion order, which is the order in which the mappings were inserted into the map
- Access order, which is the order in which the mappings were last accessed

A `LinkedHashMap` keeps an internal linked list that references mappings. If you are using the map in insertion order mode, the linked list references the mappings in the order that they were inserted. If you are using the map in access order mode, the linked list references the mappings in the order that they were accessed with either `put` or `get`.

In access order mode, the mappings that have been accessed the most are placed at the end of the linked list, and the mappings that have been accessed the least are placed at the beginning. This mode is useful in applications that need to know which data items are being accessed the most or the least.

You indicate which sorting order you prefer with the appropriate `LinkedHashMap` constructor. Table 18-12 summarizes them.

**Table 18-12** `LinkedHashMap` constructors

| | |
|---|---|
| `LinkedHashMap()` | This constructor creates an empty `LinkedHashMap` using insertion-order mode, an initial capacity of 16, and a load factor of 0.75. |
| `LinkedHashMap(int initialCapacity)` | This constructor creates an empty `LinkedHashMap` using insertion-order mode, an initial capacity of the specified amount, and a load factor of 0.75. |
| `LinkedHashMap(int initialCapacity, float loadFactor)` | This constructor creates an empty `LinkedHashMap` using insertion-order mode and the specified values for initial capacity and a load factor. |
| `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)` | This constructor creates an empty `LinkedHashMap` using the specified values for initial capacity and a load factor. If `accessOrder` is true, the map will use access-order mode. If `accessOrder` is false, the map will use insertion-order mode. |
| `LinkedHashMap(Map<? extends K, ? extends V> m)` | This constructor creates a `LinkedHashMap` and stores the mappings of the map `m` in it. The initial capacity will be at least enough to hold the elements of `m`, and the load factor will be 0.75. It throws a `NullPointerException` if `m` is a null reference. |

The program in Code Listing 18-17 is a modification of Code Listing 18-16. In this version, a `LinkedHashMap` is used in insertion-ordering mode, and the `forEach()` method is used to display the mappings. In the program output, the mappings are displayed in the order that they were inserted into the map.

**Code Listing 18-17** (`CarHashMap4.java`)

```java
import java.util.*;

/**
   This program retrieves the mappings from a HashMap
   by using the forEach method.
*/

public class CarHashMap4
{
   public static void main(String[] args)
   {
      // Create a HashMap to store Car objects.
      Map<String, Car> carMap = new LinkedHashMap<>();

      // Create some Car objects.
      Car vw = new Car("227H54", "1997 Volkswagen");
      Car mustang = new Car("448A69", "1965 Mustang");
      Car porsche = new Car("453B55", "2007 Porsche");
      Car bmw = new Car("177R60", "1980 BMW");

      // Put some mappings into the HashMap. In each
      // mapping, the car's VIN is the key and the
      // Car object containing that VIN is the value.
      carMap.put(vw.getVin(), vw);
      carMap.put(mustang.getVin(), mustang);
      carMap.put(porsche.getVin(), porsche);
      carMap.put(bmw.getVin(), bmw);

      // Iterate through the mappings, printing each one.
      System.out.println("Here are the mappings:");
      carMap.forEach( (key, value) ->
                          {
                             System.out.println("Key =" + key);
                             System.out.println("Value =" + value);
                             System.out.println();
                          });
   }
}
```

**Program Output**

```
Here are the mappings:
Key = 227H54
Value = VIN: 227H54  Description: 1997 Volkswagen

Key = 448A69
Value = VIN: 448A69  Description: 1965 Mustang

Key = 453B55
Value = VIN: 453B55  Description: 2007 Porsche

Key = 177R60
Value = VIN: 177R60  Description: 1980 BMW
```

## The `SortedMap` Interface and the `TreeMap` Class

`SortedMap<K, V>` is a subinterface of `Map<K, V>` that allows maps to store mappings by key order. The `TreeMap` class implements `SortedMap`. Table 18-13 lists some of its constructors.

**Table 18-13** `TreeMap<K, V>` constructors

| | |
|---|---|
| `TreeMap()` | This constructor creates a tree map that orders keys by their natural order. The key type `K` must implement the `Comparable` interface. |
| `TreeMap(Comparator<? super K> c)` | This constructor creates a tree map that uses the comparator `c` to order the keys. |

Using `TreeMap` is no different from using the other types of maps. For example, we can replace Line 13 of Code Listing 18-17 with:

```
SortedMap<String, Car> carMap = new TreeMap<>();
```

The program will create a tree map that will order its entries in alphabetical order of the VIN number. If we run the resulting program, the output will be as shown below.

**Program Output for `TreeMap` version of Code Listing 18-17**

```
Here are the mappings:
Key = 177R60
Value = VIN: 177R60  Description: 1980 BMW

Key = 227H54
Value = VIN: 227H54  Description: 1997 Volkswagen

Key = 448A69
Value = VIN: 448A69  Description: 1965 Mustang

Key = 453B55
Value = VIN: 453B55  Description: 2007 Porsche
```

**Checkpoint**

18.29 Each element in a map is comprised of what two parts?

18.30 What part of a mapping does the `HashMap` store by its hash code?

18.31 The `LinkedHashMap` class allows you to retrieve mappings in what two orders?

18.32 How are the keys in a `SortedMap` sorted?

## 18.5 The `Collections` Class

**CONCEPT: The `Collections` class provides numerous static methods for performing algorithms and utility operations on collections.**

The JCF has a class named `Collections` which provides numerous static methods that perform useful operations on collections. Note that the name of this class, `Collections`, is plural, and should not be confused with the `Collection` interface.

Many of the methods in the `Collections` class perform operations on lists. There are methods for copying, sorting, searching, randomly shuffling, filling with a value, and many other procedures. Table 18-14 describes several of the `Collections` methods.

**Table 18-14** Some of the static methods in the **`Collections`** class

| | |
|---|---|
| `static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)` | Performs a binary search for the element specified by `key` in the list specified by `list`.<br>Note that the elements in the list must implement the `Comparable` interface and be sorted according to their natural order. If the element is found, the method returns its index. Otherwise it returns a negative number. |
| `static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)` | Performs a binary search for the element specified by `key` in the list specified by `list`.<br>This version of the method uses the comparator `c` to make comparisons. If the element is found, the method returns its index. Otherwise, it returns a negative number. The list must be sorted in comparator order before it is passed to this method. |
| `static boolean disjoint (Collection<?> c1, Collection<?> c2)` | Returns `true` if the collections `c1` and `c2` have no elements in common. |
| `static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)` | Returns the maximum element in the collection `coll`. This method requires that all elements of `coll` implement the `Comparable` interface. There is a similar method `min` that returns the minimum. |
| `static <T> T max(Collection<? extends T>coll, Comparator<? super T> comp)` | Returns the maximum element in the collection `coll`. This method uses the comparator `comp` to compare the elements. There is a similar method `min` that returns the minimum. |
| `static void shuffle(List<?> list)` | This method rearranges the elements of list in a random order. |
| `public static <T extends Comparable<? super T>> void sort(List<T> list)` | Sorts the elements in `list` by their natural order. This method requires that the elements of list implement the `Comparable` interface. |
| `public static <T> void sort(List<T> list,Comparator<? super T> c)` | Sorts the elements in `list` according to the order imposed by the comparator `c`. |

## In the Spotlight:

### Using Anonymous Subclasses to Initialize Collections

Normally, you create anonymous classes from interfaces or abstract superclasses because you want to override their methods. You can also create an anonymous class from a concrete class and override some of its methods. In particular, you can do this with sets, lists, and maps. For example, here is the syntax for creating an anonymous subclass of `HashMap`:

```
Map<String, Integer> myMap = new HashMap<String, Integer>()
{
   // optional initialization block
   {
      // initialization code
   }
   // optional method overrides
};
```

As the example syntax shows, you can have a block of code to initialize the instance that is being created from the anonymous class. If you are instantiating a set, list, or map, you can use the initialization block to add elements. Here is an example that initializes a map and prints its contents.

```
Map<String, Integer> myMap = new HashMap<String, Integer>()
{
   {
      put("Mary", 4);
      put("Sam", 3);
   }
};
// Print the map
System.out.println(myMap);
```

## 18.6 The Stream API

**CONCEPT: A stream is an object that permits a pipeline of operations to be performed on a sequence of elements drawn from a source.**

A *stream* is an object that permits a pipeline of operations to be performed on a sequence of elements drawn from a source. The *source* of elements may be an array, a collection, or a function that is able to generate an element each time it is called. By a *pipeline* of operations, we mean a sequence of operations where the output of one operation becomes the input to the next operation in the sequence.

The operations in a stream pipeline may be of two kinds. An *intermediate* operation acts on the stream and turns it into another stream with the same source: this resulting stream may then be acted on by the next operation in the pipeline. A *terminal* operation, also called a *reduction*, acts on the stream and yields a non-stream result. A stream pipeline always begins with a source of elements, followed by zero or more intermediate operations, and ends with a single terminal operation.

## The Stream Package

The `java.util.stream` package defines a number of interfaces and classes. Among them is `Stream<T>`, a generic interface that defines the methods supported by streams whose elements are objects of type `T`. Unlike the collection API, which only supports reference types and requires primitive types to be boxed, the stream API has direct support for streams of primitive types through the `IntStream`, `DoubleStream`, and `LongStream` interfaces. There is also a `Collectors` class that has static methods for implementing useful reduction operations that accumulate stream elements into different type of collections or summary values.

## Creating a Stream

There are many ways to create a stream. The `Stream` interfaces, the `Arrays` class, the `Collection` interface, and the `Random` class, all have methods that can be used to create streams.

1. The `Arrays` class has several static `stream()`methods that take an array parameter and return a stream whose source is the array:

   ```
   static Stream<T> stream(T[] array)
   static DoubleStream stream(double[] array)
   static IntStream stream(int[] array)
   static LongStream stream(long[] array)
   ```

   The first of these methods is used to obtain a stream from an array whose elements are reference types. The other three `stream()` methods return streams of primitive types.

2. The `Collection<E>` interface defines an instance method

   ```
   Stream<E> stream()
   ```

   that returns a stream whose source is the collection.

3. All stream interfaces have a static `of( )` method with a var-args parameter that lets you create a stream from a specified list of values:

   ```
   static Stream<T> of(T ... values)
   static IntStream of(int ... values)
   static DoubleStream of(double ... values)
   static LongStream of(long ... values)
   ```

4. All `Stream` interfaces have a static `generate()` method that creates a potentially infinite stream of elements by repeatedly calling on a *supplier* function:

   ```
   static Stream<T> generate(Supplier<T> s)
   static IntStream generate(IntSupplier s)
   ```

```
static DoubleStream generate(DoubleSupplier s)
static LongStream generate(LongSupplier s)
```

5. Each of the `IntStream` and `LongStream` interfaces has two static methods that create a stream from a sequence of consecutive integers beginning at `m` and ending at `n`:

```
static IntStream range(int m, int n)              // n excluded
static IntStream rangeClosed(int m, int n)        // n included
static LongStream range(long m, long n)           // n excluded
static LongStream rangeClosed(long m, long n)     // n included
```

   The `rangeClosed()` methods include the upper bound `n`, while the `range()` methods exclude it.

6. Objects of the `java.util.Random` class have methods that can return `IntStream`, `DoubleStream`, and `LongStream` streams of random numbers. As an example, we list the following methods of `Random` that return an `IntStream` object:

```
IntStream ints()
IntStream ints(int m, int n)
IntStream ints(long count)
IntStream ints(long count, int m, int n)
```

   The `ints()` method returns a stream that can generate infinitely many random integers. The `ints(m,n)` method returns a stream that can generate infinitely many integers bounded below by `m` and above by `n`. The versions of `ints()` that take a `count` parameter are similar, except that they generate a finite stream of `count` elements.

In addition to these methods for creating a stream, the stream interfaces define many intermediate operations that transform an existing stream into another stream. We will look at those later.

## Terminal Operations

A *terminal* operation, also called a *reduction*, operates on a stream and produces a non-stream result. The result produced by a reduction is often an aggregate of some type, or a summary of some characteristics of the stream elements. Some terminal operations may perform actions on stream elements (such as printing or updating other program variables) without returning a value.

One of the simplest reduction operators is `long count()`, which returns the number of elements in a stream. The following code creates a stream of string elements `stooges` and invokes the reduction operator `long()` on it.

```
Stream<String> stooges = Stream.of("Larry", "Curly", "Moe");
System.out.println(stooges.count());
```

Table 18-15 lists some of the simpler stream reduction operators.

**Table 18-15** Stream reduction operators

| Method | Description |
|---|---|
| `long count()` | Returns the number of elements in this stream. |
| `void forEach(Consumer<? super T> action)` | Calls the specified action for each element of this stream. |
| `Optional<T> max(Comparator<? super T> c)` | Returns the maximum of all elements in this stream according to the comparator `c`. |
| `Optional<T> min(Comparator<? super T> c)` | Returns the minimum of all elements in this stream according to the comparator `c`. |
| `boolean noneMatch(Predicate<? super T> p)` | Returns `true` if no elements of this stream satisfy the predicate `p`, `false` otherwise. |
| `boolean allMatch(Predicate<? super T> p)` | Returns `true` if all elements of this stream satisfy the predicate `p`, `false` otherwise. |
| `boolean anyMatch(Predicate<? super T> p)` | Returns `true` if at least one element of this stream satisfies the predicate `p`, `false` otherwise. |
| `<A> A[] toArray(IntFunction<A[]> generator)` | Returns an array containing the elements of this stream. The array is created using a function that takes an `int` parameter `x` and returns an array of `x` elements of type `A`. |

Consider the reduction operator:

```
Optional<T> max(Comparator<? super T> c)
```

This operator requires a comparator for elements of type `T` as parameter and returns a value of type `Optional<T>`. This type represents a value of type `T` that may not exist. The maximum of all elements in a stream does not exist if the stream has *no* elements. The class `Optional<T>` has a method

```
boolean isPresent()
```

that returns true if the optional value is actually present. The class also has a method

```
T get()
```

that returns the value if it is indeed present. If `get()` is called when the value is not present, it will throw `NoSuchElementException`.

Now, suppose we want to find and print the string in the `stooges` stream that comes first in alphabetic order. We can use the statements:

```
stooges = Stream.of("Larry", "Curly", "Moe");
Optional<String> min = stooges.min( (x, y) -> x.compareTo(y) );
System.out.printf("First stooge by alphabetic order: %s\n", min.get());
```

Here we have used the lambda expression

```
(String x, String y) -> s.compareTo(y)
```

to create an object that implements the `Comparator` interface for string types. It turns out that Java 8 allows a shorter way to write such a lambda expression: you can just write `String::compareTo`. The call to the `min` reduction can therefore be written like this:

```
Optional<String> min1 = stooges.min(String::compareTo);
```

In the code above, we do not bother to call `min.isPresent()` before we call `min.get()` because we know our `stooges` stream is not empty. You should call `isPresent()` before calling `get()` whenever there is a possibility that the stream is empty.

The reduction operator

```
<A> A[] toArray(IntFunction<A[]> generator)
```

collects stream elements into an array that is created by an array *generator*. This is a function that takes an integer `x` and returns an array of length `x`, whose elements have type `A`. In our case, the type `A` is `String`, so we want the lambda expression:

```
x -> new String [x]
```

Again, Java 8 provides a convenient notation for such a function: simply write `String[]::new`. In general, for any type `A`, the expression `A[]::new` represents a function that takes an integer parameter `x` and returns an array of type `A` that has `x` elements.

Code Listing 18-18 illustrates how to use some of the reduction operators in Table 18-15.

**Code Listing 18-18** (`StreamReductionsDemo.java`)

```
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;
/**
 This program illustrates the use of a
 number of stream reduction operators.
*/

public class StreamReductionsDemo
{
    public static void main(String[] args)
    {
        // Create the source.
        String[] stoogesArray0 =
        {
            "Larry", "Curly", "Moe"
        };
        System.out.printf("The Stream source is the array: %s \n",
                Arrays.toString(stoogesArray0));
        // Variable for the stream.
        Stream<String> stooges;

        // Get a stream and reduce with count.
        stooges = Stream.of(stoogesArray0);
        System.out.printf("The count is %d\n", stooges.count());

        // Get a stream and reduce with forEach.
        stooges = Stream.of(stoogesArray0);
        stooges.forEach(x ->
        {
            System.out.printf("%s has length %d\n", x, x.length());
        });

        System.out.println("The stream reduced to an array (2 ways):");

        // Get a stream and reduce with toArray().
        stooges = Stream.of(stoogesArray0);
        String[] stoogesArray1 = stooges.toArray(x -> new String[x]);
        System.out.println(Arrays.toString(stoogesArray1));

        // Get a stream and reduce with toArray().
        stooges = Stream.of(stoogesArray0);
        String[] stoogesArray2 = stooges.toArray(String[]::new);
        System.out.println(Arrays.toString(stoogesArray2));

        // Get a stream and reduce with min.
        stooges = Stream.of(stoogesArray0);
        Optional<String> min = stooges.min((x, y) -> x.compareTo(y));
        System.out.printf("First stooge by alphabetic order: %s\n",
                                                          min.get());
        // Get a stream and reduce with min.
        stooges = Stream.of(stoogesArray0);
        Optional<String> min1 = stooges.min(String::compareTo);
        System.out.printf("First stooge by alphabetic order: %s\n",
                                                          min1.get());
        // Get a stream and reduce with max.
        stooges = Stream.of(stoogesArray0);
        Optional<String> max = stooges.max(String::compareTo);
        System.out.printf("Last stooge by alphabetic order: %s\n",
                                                          max.get());
    }
}
```

**Program Output**

```
The Stream source is the array: [Larry, Curly, Moe]
The count is 3
Larry has length 5
```

```
Curly has length 5
Moe has length 3
The stream reduced to an array (2 ways):
[Larry, Curly, Moe]
[Larry, Curly, Moe]
First stooge by alphabetic order: Curly
First stooge by alphabetic order: Curly
Last stooge by alphabetic order: Moe
```

In Code Listing 18-18, we have to recreate the stream from the source every time we execute a reduction operation on it:

```
stooges = Stream.of(stoogesArray0);
```

This is because a stream object can be reduced only one time. Once it is reduced, the stream is drained of all its elements and becomes unuseable.

The `IntStream`, `DoubleStream`, and `LongStream` have additional reduction operators. Because of the close similarity, we will list only the operators for `DoubleStream`. Analogous operators exist for the other two streams.

```
OptionalDouble average()
OptionalDouble max()
OptionalDouble min()
Double sum()
double[] toArray()
```

These operators respectively reduce a stream of numbers of type `double` to their average, maximum, minimum, and sum. The `max()` and `min()` are similar to the corresponding operators of the same name in `Stream<T>`, but do not require a comparator because the natural order of numbers is used. The last operator collects stream elements into an array of type `double`.

## Intermediate Operations on Streams

An intermediate operation acts on an existing stream and produces a new stream. Table 18-16 lists some of the intermediate operations defined in the `Stream<T>` interface.

**Table 18-16** `Stream<T>` intermediate operators

| | |
|---|---|
| `Stream<T> distinct()` | Removes duplicates and returns the resulting stream. |
| `Stream<T> filter(Predicate<? super T> predicate)` | Returns a stream consisting of only those elements that satisfy the given predicate. |
| `Stream<T> limit(long n)` | Returns a stream consisting of at most the first `n` elements of this stream. |
| `<R> Stream<R> map(Function<? super T,? extends R> mappingFunction)` | Returns a stream whose elements are the result of calling the mapping function on elements of this stream. |

*(table continues next page)*

**Table 18-16** `Stream<T>` intermediate operators (continued)

| | |
|---|---|
| `Stream<T> peek(Consumer<? super T> action)` | Returns a stream consisting of the same elements as this stream. The given action method will be called on each element of the resulting stream as the element is being drawn from the stream. |
| `Stream<T> skip(long n)` | Returns this stream after discarding the first `n` elements. |
| `Stream<T> sorted()` | Returns a stream with the same elements as this stream, but sorted according to their natural order. |
| `Stream<T> sorted(Comparator <? super T> comparator)` | Returns a stream with the same elements as this stream, but sorted according to the given comparator. |

Code Listing 18-19 illustrates some of these intermediate operators. The program creates a stream that is able to produce infinitely many random integers less than 10. A filter operation is then plugged onto that stream to filter for even numbers, followed by a `limit(15)` operation that will ask for only 15 of those numbers. The last operation in the pipeline reduces the resulting stream to an array, which is then printed.

**Code Listing 18-19** (`StreamIntermediateOpsDemo.java`)

```
import java.util.Arrays;
import java.util.Random;
import java.util.stream.IntStream;

/**
   This program illustrates intermediate Stream operations.
*/

public class StreamIntermediateOpsDemo
{
    public static void main(String [] args)
    {
        // Random number generator.
        Random randy = new Random();

        // Potentially infinite stream of random
        // numbers all of which are less than 10.
        IntStream infRandom = randy.ints(0, 10);

        // Filter out all the odd numbers, keep only even numbers.
        IntStream intEvenRandom = infRandom.filter(x -> x % 2 == 0);

        // Keep only the first 15 of the infinite
        // stream of random even numbers.
        IntStream first15 = intEvenRandom.limit(15);

        // Reduce to an array and print.
        int [] evenRandomArr = first15.toArray();
        System.out.println("The result of stream operations is: ");
        System.out.println(Arrays.toString(evenRandomArr));
    }
}
```

**Program Output**

```
The result of stream operations is:
[0, 2, 6, 2, 2, 6, 8, 4, 8, 8, 6, 4, 6, 6, 4]
```

## Intermediate Stream Operations are Lazy

You may well be wondering how the stream

```
randy.ints(0,10)
```

can represent an infinite stream of numbers as claimed. The reason is that all intermediate stream operations are *lazy*: the intermediate operations will produce an element only when the element is being demanded by a reduction operation. To see how this works, let us remove all the "temporary" variables that are keeping track of the streams at intermediate stages. We end up with the pipeline expression:

```
evenRandomArr = randy.ints(0, 10)
                     .filter(x -> x% 2 == 0)
                     .limit(15)
                     .toArray();
```

When this is executed, the `toArray()` reduction demands all elements in the aggregate stream.

The `limit(15)` operation will then demand 15 elements from the filter operation, which will keep demanding elements from `int(0,10)` until it has passed 15 even numbers to `limits(15)`. At that point, `limits(15)` is satisfied and stops asking for elements from filter, which stops asking for elements from `ints(0,10)`.

You can think of a stream as a conduit through which objects flow, beginning at a source and ending at some terminal operation. The `peek()` operation executes an action on each element as it passes by. Consider Code Listing 18-20, and the output that follows the code listing. We have modified the stream pipeline shown above by placing a `peek()` on elements as they leave the `ints(0,10)` stream, and another `peek()` as the elements leave the `filter()`. To cut down on the volume of the output, we limit the size of the final stream to 5 elements.

**Code Listing 18-20** (`PeekDemo.java`)

```
import java.util.Arrays;
import java.util.Random;
import java.util.stream.IntStream;

```

```
/**
  This program illustrates the working of the
  peek intermediate operation.
*/

public class PeekOpDemo
{
    public static void main(String [] args)
    {
        // Random number generator.
        Random randy = new Random();

        // Potentially infinite stream of random
        // numbers all of which are less than 10.
        IntStream infRandom = randy.ints(0, 10);

        int [] evenRandomArr =
                randy.ints(0, 10)
                     .peek(x -> {System.out.printf("ints: %d\n", x);})
                     .filter(x -> x% 2 == 0)
                     .peek(x-> {System.out.printf("filter: %d\n", x);})
                     .limit(5)
                     .toArray();
        System.out.println("The result of stream operations is: ");
        System.out.println(Arrays.toString(evenRandomArr));
    }
}
```

**Program Output**

```
ints: 9
ints: 0
filter: 0
ints: 9
ints: 1
ints: 5
ints: 6
filter: 6
ints: 3
ints: 9
ints: 4
filter: 4
ints: 4
filter: 4
ints: 6
filter: 6
The result of stream operations is:
[0, 6, 4, 4, 6]
```

Notice that `ints()` produces 11 elements altogether, 6 odd and 5 even. All odd elements make it out of the `int()` stream, but get filtered out by `filter`().

## The Stream `map` Operation

The `map` operation

```
<R> Stream<R> map(Function<? super T,? extends R> mappingFunction)
```

applies a function of type $T \rightarrow R$ to transform a stream of elements of type $T$ into a stream of type $R$. Actually the input type of the function needs only be a superclass of $T$, and the return type only needs to be a subclass of $R$.

A simple example of this is the following. If you have a stream of consecutive integers

```
[1,  2,  3,  4,  5,  6]
```

you can apply the mapping function `x -> 2*x-1` to every element of the stream and get the stream of consecutive odd numbers

```
[1,  3,  5,  7,  9,  11]
```

The code for this is:

```
int n = sc.nextInt();
IntStream odds = IntStream.rangeClosed(1, n).map(x -> 2*x -1);
System.out.println(Arrays.toString(odds.toArray()));
```

Now, suppose you want to turn a stream of strings into a string of integers by mapping each string to its length. The map operation $T \rightarrow R$ requires that $R$ be a reference type. If you use `map`, the compiler will automatically box the integer length of each string, creating a stream of type `Stream<Integer>` instead of one of type `IntStream`.

```
Stream<String> names = Stream.of("Abby", "Bob", "Christopher",
                                         "David", "Elizabeth");
Stream<Integer> lengths = names.map(x -> x.length());
```

This means that executing the lines of code above will result in lengths being a stream of `Integer` rather than `int` elements. If you want to map a `Stream<T>` object to a stream whose elements are primitive types, you need to use the following `mapToXXX()` methods of the `Stream<T>` interface.

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
IntStream mapToInt(ToIntFunction<? super T> mapper)
LongStream mapToLong(ToLongFunction<? super T> mapper)
```

These methods take as input a function that takes a parameter of type $T$ and returns a `double`, an `int`, or `long` value. The proper way to map a stream of strings to an `IntStream` is like this:

```
Stream<String> names = Stream.of("Abby", "Bob", "Christopher",
                                           "David", "Elizabeth");
IntStream lengths = names.mapToInt(x -> x.length());
System.out.println(Arrays.toString(lengths.toArray()));
```

## The Stream `reduce` Methods

Many reduction operations can be viewed as a process of "accumulating" stream elements into a single value of the same type. The simplest example of this is when the elements of a stream of integers are processed to yield another integer that is the sum of the stream elements. Another example is the processing of a stream of integers to yield a product.

To perform an accumulation reduction on a stream of type $T$, you need an *accumulation operator* $p$ of type $(T, T) \rightarrow T$ that will be used to combine two values of type $T$ into a single value. You also need a value $i$ of type $T$ to represent the result of performing the accumulation reduction on an empty stream. This value is called the *identity* value for $p$ and must satisfy the relation $p(i, x) = x$ for all values $x$ of type $T$. The accumulation process uses an internal variable $a$ to represent the value of stream elements already accumulated. Initially, no elements have been accumulated and $a$ is set to $i$. The process proceeds by setting $a = p(a, x)$ for each stream element $x$. The final value of $a$ is returned at the end of the process.

As an example, suppose we want to reduce a stream of integers to the sum of its elements. The accumulation operator $p$ is a function that returns the sum of two integers. For example, $p$ might be the lambda expression `(x,y)-> x+y`. The identity value for addition is the integer zero. Initially, the internal variable $a$ would be 0. At each stage of the reduction, the current stream element would be added to $a$ by setting $a = a + x$.

All three interfaces for streams of primitive types define `reduce()` methods that implement accumulation reductions of this nature:

```
int reduce(int identity, IntBinaryOperator op)
long reduce(long identity, LongBinaryOperator op)
double reduce(double identity, DoubleBinaryOperator op)
```

When you are accumulating an integer stream to find the sum, you use `(x,y)-> x+y` for the binary operator and take 0 for the identity. However, if you are accumulating by multiplying, the appropriate binary operator is `(x,y)-> x*y` and the identity value is 1. The following code uses reductions to find the sum of the first 4 positive odd integers, and also the product of the first 4 positive integers.

```
int n = 4;
// Create streams of integers.
IntStream odds = IntStream.rangeClosed(1, n).map(x-> 2*x -1);
IntStream ints = IntStream.rangeClosed(1, n);

// Reduce the streams.
int oddsSum = odds.reduce(0, (x,y)->x+y );
int intsProduct = ints.reduce(1, (x, y)-> x*y);

// Print the results.
System.out.printf("The sum is %d and the product is %d\n",
                  oddsSum, intsProduct);
```

When executed, this code will print 14 for the sum and 24 for the product.

The `Stream<T>` interface has a similar `reduce()` method:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

This method works similarly to the ones we have already discussed.

The `reduce()` methods require the binary accumulator operator $p$ to be associative. A binary operator $p$ is *associative* if for any sequence of values $a$, $b$, $c$, the two values $p(a, p(b, c))$ and $p(p(a, b), c)$ are equal. The requirement allows for the parallelization of the reduction operation. The implementation may split the stream into segments, reduce these segments in parallel, and later combine the result. The associativity of the accumulator operator ensures that final result will not be affected by the order in which the results are combined.

There are times when you want to reduce the elements of a stream of type $T$ to a single value of a different type $U$. To meet this need, the `Stream<T>` interface defines a method:

```
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
                                        BinaryOperator<U> combiner)
```

This method requires an *accumulator* bi-function $f$ of type $(U, T) \rightarrow U$ to "add" a current value of type $T$ to the accumulated value of type $U$. It also requires a value $i$ of type $U$ to act as the initial accumulation value to get the process started. At each stream element $x$, the reduction replaces the current accumulation value $a$ with $f(a, x)$ by setting $a = f(a, x)$. The reduction may split the stream into segments and reduce the segments in parallel, with each segment producing an accumulated value of type $U$. These values are combined two at a time by using a binary *combiner* operator $c$, which must be associative. Moreover, the accumulation value $i$ must be an *identity* value for the operator $c$.

As an example, suppose that we want to reduce a stream of `String` elements by collecting them into a set. We will see shortly that there is a stream `Collectors` method that will do that, but let us push ahead for the sake of illustration. Here the type $T$ is `String`, while the type $U$ is `Set`. At each stage, the accumulated value will be a set, and the accumulator bi-function will be a function that takes a set and a string, adds the string to the set, and returns the resulting set:

```
(Set s, String x) -> { s.add(x); return s; }
```

The combiner operator will take two sets, add all the elements of the first set to the second set, and return the second set:

```
(Set s, Set t) -> { t.addAll(s); return t; }
```

Finally, the identity value that gets the set-building process started is an empty set:

```
new HashSet<String>();
```

Code Listing 18-21 shows the complete program.

**Code Listing 18-21** (`ReduceToSetDemo.java`)

```
import java.util.HashSet;
import java.util.Set;
import java.util.function.BiFunction;
import java.util.function.BinaryOperator;
import java.util.stream.Stream;

```

```java
/**
  This program demonstrates how to use the Stream
  reduce method.
*/

public class ReducetoSetDemo
{
    public static void main(String [] args)
    {
        Stream<String> stooges = Stream.of("Larry", "Curly", "Moe");

        // The accumulator for the reduction.
        BiFunction<Set<String>, String, Set<String>> accumulator =
               (s, x) -> { s.add(x); return s;};

        // The combiner for the reduction
        BinaryOperator<Set<String>> combiner =
                (s, t) -> { t.addAll(s); return t;};

        // Reduce the stream to a set.
        Set<String> stoogeSet = stooges.reduce(new HashSet<String>(),
                                               accumulator, combiner);
        // Print the set
        System.out.printf("The set of stooges is %s\n", stoogeSet);
    }
}
```

**Program Output**

```
The set of stooges is [Moe, Larry, Curly]
```

Notice, we can easily convert a stream into a set using the `forFach()` reduction method that we learned about earlier. The advantage of the `reduce()` methods lies in the potential for parallelization. When working with large quantities of data, parallelization may result in significant savings in time.

## Stream Collectors

Collectors are a slightly different approach to reductions. *Collectors* are typically used to accumulate stream elements into a collection, although they can perform other types of reductions.

The `Stream<T>` interface has a method

```java
<R,A> R collect(Collector<? super T,A,R> collector)
```

that uses a collector object to reduce a stream of elements of type *T* into a single value of type *R*. The collector accumulates stream elements into a value of type *A*, and then transforms the type *A* value into a final result of type *R* when all stream elements have been processed.

The `Stream` package has a `Collectors` class that provides static methods that return different types of collectors. Among them are the two methods:

```
static <T> Collector<T,?,List<T>>  toList()
static <T> Collector<T,?,Set<T>>  toSet()
```

The `toList()` method returns a collector that will reduce a stream of elements of type *T* by collecting them into a list of type `List<T>`. Here the final type *R* is `List<T>`, and the intermediate type *A* is an unspecified implementation detail. This is because the user of the collector does not need to know the internal type used for the intermediate accumulated values. The `toSet()` method is similar.

Let us illustrate the use of the `toSet()` collector. The following code collects the elements of a stream of strings into a set and prints the set:

```
Stream<String> stooges = Stream.of("Larry", "Curly", "Moe");
Set<String> stoogesSet = stooges.collect(Collectors.toSet());
System.out.printf("The set of stooges is %s\n", stoogesSet);
```

As you can see, the `collect` method takes a lot less code than the program of Code Listing 18-21.

Another interesting method is `toMap()`:

```
static <T,K,U> Collector<T,?,Map<K,U>>
    toMap(
           Function<? super T,? extends K> keyMapper,
           Function<? super T,? extends U> valueMapper
          )
```

This method returns a collector that can reduce a stream to a map. The method takes as parameters two functions: a key mapper that determines the key for a stream element, and a value mapper that determines the value associated with the key. It is these key-value pairs that are stored in the map.

To illustrate, suppose we have a stream of strings, and we want to create a map that stores the strings as values, with each string being keyed by its length. For example, a stream of the strings

```
"Larry", "Moe", "Al"
```

would result in a map that stores the following key-value pairs:

```
(5, Larry)
(3, Moe)
(2, Al)
```

In the syntax of the definition of `toMap()`, the stream element type *T* is `String`, the key type *K* is `Integer`, and the value type *U* is `String`. Our key-mapping function is the lambda expression,

```
(String x)-> x.length()
```

while the value-mapping function is the lambda expression

```
(String x) -> x
```

A program that reduces a stream of strings to such a map is shown in Code Listing 18-22.

**Code Listing 18-22** `(ReduceToMapDemo1.java)`

```java
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
  This program demonstrates the Collectors.toMap method.
*/

public class ReduceToMapDemo1
{
    public static void main(String [] args)
    {
        // This is the stream to reduce.
        Stream<String> names = Stream.of("Larry", "Moe", "Al");

        // key of a string is its length.
        Function<String, Integer> keyMapper = x -> x.length();

        // The value of a string is the string itself.
        Function<String, String> valueMapper = x -> x;

        // Reduce the stream to a map
        Map<Integer, String> namesMap =
           names.collect(Collectors.toMap(keyMapper, valueMapper));

        // Print the map.
        System.out.printf("The map of stooges is %s\n", namesMap);
    }
}
```

**Program Output**

```
The map of stooges is {2=Al, 3=Moe, 5=Larry}
```

In the input to the program of Code Listing 18-22, there is at most one string with a given length. This is because the lengths of the strings are keys of a map, and maps require keys to be unique. A more realistic reduction would allow any number of strings of a given length and groups the strings according to their length. Such a reduction would reduce a stream of strings to a map whose values are lists of strings, and the key of a value is the common length of all strings in the list. For example, the stream of strings

```
"Larry", "Curly", "Moe", "Al"
```

would be reduced to the map

```
(5, [Larry, Curly])
(3, [Moe])
(2, [Al])
```

The `Streams<T>` interface has such a method:

```
static <T,K,U> Collector<T,?,Map<K,U>>
    toMap(
            Function<? super T,? extends K> keyMapper,
            Function<? super T,? extends U> valueMapper,
            BinaryOperator<U> mergeFunction
         )
```

This method is just like the last one, except it has a *merge function* parameter that is a binary operator on the type $U$. The merge function $m$ has type $(U, U) \rightarrow U$, meaning that $m$ takes two parameters of type $U$ and returns a single result of the same type. If two values $u_1$ and $u_2$ have the same key $k$, the `toMap()` method will call the merge function on the two values and use the result $m(u_1, u_2)$ as the "merged" value for $k$.

Let's look at how this plays out if we want to reduce a stream of strings into a map that groups strings by their lengths. First, note that the type $U$ of the final result is no longer `String`, but `List<String>`. The key mapper is the same as before, but the value mapper now assigns to each string $x$ a list whose only member is $x$:

```
Function<String, List<String>> valueMapper = x->
    {
        List<String> listOfOne = new ArrayList<>();
        listOfOne.add(x);
        return listOfOne;
    };
```

The merge function takes two lists that would have the same key and returns a single list that is the concatenation of the two lists:

```
BinaryOperator<List<String>> mergeFunction = (list1, list2) ->
    {
        list1.addAll(list2);
        return list1;
    };
```

Code Listing 18-23 shows a complete program that implements the reduction.

**Code Listing 18-23** (`ReduceToDemo2.java`)

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**

  This program demonstrates the Collectors.toMap method.
*/

public class ReduceToMapDemo2
{
    public static void main(String [] args)
    {
        // This is the stream to reduce to a map.
        Stream<String> names = Stream.of("Larry", "Curly", "Moe", "Al");

        // The key for a string is the length of the string.
        Function<String, Integer> keyMapper  = x -> x.length();

        // The value for a string is a list of all strings in
        // the stream with the same length. Every string is initially
        // mapped to list containing just that string.
        Function<String, List<String>> valueMapper =  x->
        {
            List<String> listOfOne = new ArrayList<>();
            listOfOne.add(x);
            return listOfOne;
        };

        // When two values have the same key, the merge function
        // is used to merge the two values and the merged result
        // becomes the value of the key.
        BinaryOperator<List<String>> mergeFunction = (list1, list2) ->
        {
            list1.addAll(list2);
            return list1;
        };

        //Reduce the stream and get a map.
        Map<Integer, List<String>> namesMap =
        names.collect(Collectors.toMap(
                                        keyMapper, valueMapper,
                                        mergeFunction));

        // Print the map.
        System.out.printf("The map of names is %s\n", namesMap);
    }
}
```

**Program Output**

```
The map of names is {2=[Al], 3=[Moe], 5=[Larry, Curly]}
```

The `Stream<T>` interface has a second `collect` method that takes as parameters the ingredients needed to build a collector. The method internally builds the collector and uses it to reduce the stream. You can use this method in the event that none of the collectors in the `Collectors` class suit your needs.

```
<R> R collect(
               Supplier<R> supplier,
               BiConsumer<R,? super T> accumulator,
               BiConsumer<R,R> combiner
             )
```

Briefly, the supplier is a method that is called to provide an initial value of the final type *R*; the role of the supplier is similar to the role of the identity in the reduce methods. The accumulator and combiners play roles similar to those of the accumulator and combiner in the reduce methods. You can learn more about this and other methods by studying the streams API documentation.

**Checkpoint**

18.33 How do you define a stream of elements?

18.34 How does a stream intermediate operation differ from a terminal operation?

18.35 Name at least two different classes or interfaces that have methods for creating streams.

18.36 What is a stream pipeline?

18.37 What is the difference between a terminal operation and a reduction?

18.38 What is a collector?

18.39 How many reduction operations can be in a stream pipeline?

## 18.7 Common Errors to Avoid

- **Not calling an iterator's `hasNext` method before calling `next`.** To prevent an exception from being thrown, you should always call an iterator's `hasNext` method before calling its `next` method. This ensures that the `next` method will actually return a reference.
- **Not calling an iterator's `next` method before calling `remove`.** An iterator's `remove` method removes the last element that was returned by the `next` method. The `next` method must be called before the `remove` method can be called.
- **Not calling a `ListIterator`'s `hasPrevious` method before calling `previous`.** To prevent an exception from being thrown, you should always call a `ListIterator`'s `hasPrevious` method before calling its `previous` method.
- **Not calling a `ListIterator`'s `hasNext` or `hasPrevious` method immediately before calling `set`.** To prevent an exception from being thrown, you should always call a `ListIterator`'s `hasNext` or `hasPrevious` method before calling its set method.
- **Modifying a collection via one of its methods after an iterator has been created for the collection.** After an iterator has been created for a collection, you cannot modify the collection using one of the collection's methods and then continue to use the iterator. Doing so will cause an exception to be thrown. If you wish to modify a collection while

it is being iterated over, you should use the iterator to perform the modifications. For example, the `Iterator` interface specifies a remove method, and the `ListIterator` interface specifies `add` and `set` methods.

- **Designing a class so objects of the class can be stored in a `HashSet`, but not overriding both the `equals` and `hashCode` methods.** When an object is stored in a `HashSet`, both the `hashCode` and the `equals` methods are used to find a location for the object in the `HashSet`. If you are designing a class so objects of the class can be stored in a `HashSet`, you should override both the `hashCode` and the `equals` methods.
- **Storing objects that cannot be compared in a `TreeSet`.** In a `TreeSet`, objects are compared using the `compareTo` method. An error will occur if you attempt to store in a `TreeSet` an object of a class that does not implement the `compareTo` method.
- **Attempting to store duplicate keys in a map.** The keys in a map must be unique.
- **Trying to reuse a stream after it has been reduced.** A stream becomes drained of its elements when it is reduced. Trying to use it after that will result in an error.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This type of collection stores elements in an ordered sequence, and allows duplicates.
   a. list
   b. set
   c. map
   d. hash set
2. Each element in one of these is comprised of two parts: a key and a value.
   a. list
   b. set
   c. map
   d. hash set
3. This type of collection is optimized for searching, stores elements in an unordered sequence, and does not allow duplicates.
   a. list
   b. set
   c. map
   d. `ArrayList`
4. This interface specifies the basic operations of a list.
   a. `ListCollection`
   b. `List`
   c. `Set`
   d. `ArrayList`
5. A terminal operation in a stream pipeline is also called
   a. the stream end
   b. a reduction
   c. the sink
   d. an iterator

6. This is an object that can be used to retrieve the elements in a collection, one after the other.
    a. iterator
    b. manipulator
    c. comparator
    d. concrete collection class
7. `ListIterator` is a subinterface of
    a. `Iterator`
    b. `Collection`
    c. `Comparator`
    d. `List`
8. This `ListIterator` method replaces an existing element with a new element.
    a. `replace`
    b. `update`
    c. `set`
    d. `add`
9. A stream collector is an object that
    a. implements the `Collection` interface
    b. implements the `Map` interface
    c. is used to reduce a stream
    d. uses lambda expressions
10. This list class stores its elements by connecting each element to its successor.
    a. `ArrayList`
    b. `LinkedList`
    c. `HashList`
    d. `ConnectedList`
11. This is an object that can compare two other objects and determine whether one is greater than, less than, or equal to the other.
    a. `Comparator`
    b. `Comparer`
    c. `Sorter`
    d. `Sequencer`
12. This class provides numerous static methods that perform useful operations on collections.
    a. `Collection`
    b. `CollectionUtilities`
    c. `Collections`
    d. `Util`
13. This JCF interface is not a subinterface of `Collection`.
    a. `List`
    b. `Set`
    c. `Map`
    d. `SortedSet`
14. This class is a good choice to use when your application is storing data in a list and is doing lots of insertions and deletions in the middle of the list.
    a. `ArrayList`
    b. `Vector`
    c. `LargeList`
    d. `LinkedList`

15. **True or False:** Duplicates can be stored in a list.
16. **True or False:** Duplicates can be stored in a set.
17. **True or False:** `Collection` is a subinterface of `List`.
18. **True or False:** A `ListIterator` can move backward through a list.
19. **True or False:** A `ListIterator` can be used to modify the contents of a list.
20. **True or False:** When the Java compiler processes an enhanced `for` loop, it actually converts it to a traditional loop that uses an iterator.
21. **True or False:** You can randomly access the elements in a linked list.
22. **True or False:** When you iterate through a `HashSet`, the objects will be retrieved in the same order `that they were added to the `HashSet`.

### Find the Error

1.
   ```
   List<String> nameList = new LinkedHashSet<>();
   ```
2.
   ```
   List<String> nameList = new ArrayList<>();
   ListIterator it = new ListIterator(nameList);
   ```
3.
   ```
   Set<String> nameSet = new HashSet<>();
   nameSet.add("Chris");
   nameSet.add("Kenny");
   ListIterator it = nameSet.listIterator();
   ```
4.
   ```
   List<String> nameList = new ArrayList<>();
   ListIterator it = new ListIterator(nameList);
   it.set("Herbert");
   ```

### Algorithm Workbench

1. Write a statement that declares a `List` reference variable, instantiates an `ArrayList` object that can hold `String` objects, and assigns the address of the `ArrayList` object to the reference variable.
2. Assume that `myList` references a `List` object. Write a statement that gets a list iterator from the `myList` object.
3. Assume that `it` references a newly created iterator for a list of `String` objects. Write code that uses the iterator to display each of the `String` objects in the list.
4. In pseudocode, outline the procedure followed by a `HashSet` object to add a new object.

### Short Answer

1. What do you call an object that provides the elements that make up a stream?
2. What is the essential difference between a stream intermediate operation and a stream terminal operation?
3. What is the benefit of using an interface variable to reference a collection object?
4. What is an iterator?
5. How do you get an iterator for a list?
6. Which element does the `ListIterator` method set replace in a list?

7. How does the Java compiler process an enhanced `for` loop?
8. Which is more efficient—inserting an element in the middle of an `ArrayList`, or inserting a node into the middle of a `LinkedList`?
9. Why do we need `IntStream` when we have `Stream<Integer>`?
10. What is the purpose of the `map()` method in the `Stream<T>` interface?
11. Look at the following code:

    ```
    String str1 = new String("Barney");
    String str2 = new String("Barney");
    ```

    Will the methods `str1.hashCode()` and `str2.hashCode()` return the same value?
12. In hashing, what is a collision?
13. How does a map work?
14. What are the three guidelines that you should follow when overriding the `hashCode` and `equals` methods in a class?

## Programming Challenges

### 1. Word Set

Write an application that reads a line of input from the keyboard, and then displays each unique word that was entered, sorted in ascending order. You can do this by tokenizing the line of input and adding each token to an appropriate `Set` object.

### 2. Instructor Set

Chapter 8 presented an `Instructor` class that holds information about an instructor. (See Section 8.7—Aggregation) Modify the `Instructor` class by overriding the `hashCode` and `equals` methods. Then write a class that stores several `Instructor` objects in a `HashSet`. The class should be able to display all the instructors in the set, and allow the user to search for an instructor. Demonstrate the class in an application.

### 3. `EmployeeMap` Class

Create an `Employee` class that stores an employee's ID number and name. Then create an `EmployeeMap` class that allows you to add `Employee` objects and look them up by their ID numbers. The `EmployeeMap` class should use a `Map` object to map ID numbers to `Employee` objects. Create an application to demonstrate the classes.

### 4. Stock Map

Chapter 8 presented a `Stock` class that holds information about a stock. (See Section 8.4—The `toString` Method) Write a class that keeps `Stock` objects in a `Map`. The class should be able to retrieve a particular `Stock` object from the `Map` by searching on its stock symbol. Demonstrate the class in an application.

### 5. Prime Number List

A prime number is an integer (other than 1) that is only evenly divisible by itself and 1. For example, the number 5 is prime because it can only be evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6. Write an application that calculates and stores the first 100 prime numbers (starting at 2) in a list. Use an iterator to display the numbers.

### 6. Dealing Cards

Write a class named `Card`, which can be used to represent a card from a deck of cards. The class should be able to store a card's suit and face value. A card's suit can be one of the following: Hearts, Diamonds, Spades, or Clubs. A card's face value can be Ace, Jack, Queen, King, or a value in the range of two through ten.

Next write a `Deck` class. This class constructor should create a list of 52 `Card` objects, each representing a valid card in a deck of cards. The class should have a `shuffle` method that randomly shuffles the `Card` objects in the list. It should also have a `deal` method that "deals" a card from the deck. It does this by removing the `Card` object at the beginning of the list and returning a reference to that object.

Next, write `CardPlayer` class. This class should keep a list of `Card` objects that have been dealt to it. This represents a hand of cards. A method named `getCard` should accept a reference to a `Card` object, which is added to the list. A method named `showCards` displays the `Card` objects in the list.

Demonstrate these classes in an application that creates a `Deck` object, shuffles the cards it contains, and deals five cards from the `Deck` to a `CardPlayer` object. The `CardPlayer` should then display its cards.

### 7. Word Frequency Count

Write a program that allows the user to specify a text file, opens the file, and prints a two-column table consisting of all the words in the file together with the number of times that each word appears. Words are space-delimited and case-sensitive. The table should list the words in alphabetical order.

### 8. groupingBy Collector

Consult the online Stream API documentation and read about the `groupingBy()` method of the `Collectors` class, and use that method to get a simpler and shorter solution to the problem of classifying strings that was solved in the program of Code Listing 18-23.

# CHAPTER 19 Linked Lists

TOPICS

19.1 Introduction to Linked Lists
19.2 Operations on Linked Lists
19.3 Doubly Linked and Circularly Linked Lists
19.4 Recursion on Linked Lists
19.5 Common Errors to Avoid

## 19.1 Introduction to Linked Lists

**CONCEPT: A linked list is an implementation of a list in which elements are stored in nodes, and each node contains a link to its successor.**

As you learned in Chapter 18, a list is a collection that stores its elements in a position-based sequence. Each element in a list is associated with an index that corresponds to its position within the list. The first element has index 0, the second element has index 1, and so forth.

This position-based order gives rise to the notion of a predecessor and a successor for each element: if $X$ is a list element with index $k$, then its *successor* is the element with index $k + 1$ and its *predecessor* is the element with index $k - 1$. The first element in the list has no predecessor and the last element in the list has no successor.

There are two ways to allocate storage for the elements of a list: contiguous allocation, and linked allocation. Array-based lists use *contiguous allocation*, which allocates storage for successive list elements in consecutive memory locations. A drawback of contiguous allocation is the necessity to move large numbers of elements when inserting elements into, or deleting elements from, the middle of the list.

*Linked allocation* does not need consecutive memory locations to store list elements. Instead, it stores with each element, a pointer to the memory location of its successor. A *linked list* is a list that uses linked allocation. In Java, a linked list is implemented as a sequence of *nodes*, where each node stores a list element and a reference to the next node in the list. For example, to implement a linked list of `String` elements, we can use a class such as the following:

```
class Node
{
   String value;        // Value of a list element
   Node next;           // Link to next node in the list

   Node(String val, Node n)
   {
      value = val;
      next = n;
   }
   Node(String val)
   {
      value = val;
      next = null;
   }
}
```

A linked list is represented by a reference to its first node. The first element of the list is called the *head* of the list, so this reference is often called the *head reference*. An empty linked list is indicated by setting its head reference to `null`.

Our examples in this chapter will involve linked lists of strings. Working with linked lists of other types of elements, and even with generic linked lists, is similar.

## Creating Lists and Adding Nodes

Once we have a suitable node class, we can create linked lists by creating nodes and setting their successor links to reflect the order that the list imposes on its elements. A sequence of statements that builds a linked list is shown in Figure 19-1. In the figure, a `Node` object is shown as a rectangular box with a compartment for the element and a second compartment for the successor reference `next`.

The first statement in Figure 19-1

```
Node myList = new Node("Bob");
```

creates a linked list with the single element `"Bob"`. At this point, `myList` refers to a `Node` object that contains a field `myList.next`, which can itself be made to point to another node:

```
myList.next = new Node("Carol");
```

This creates a linked list whose two elements form the sequence `"Bob Carol"`. The newly added node contains its own `next` field, which can be accessed by using the expression `myList.next.next` and made to point to yet another node:

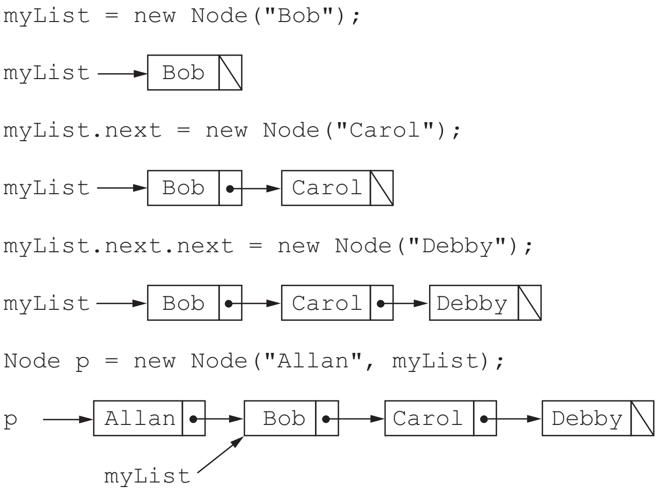
```
myList.next.next = new Node("Debby");
```

We are not limited to the end of a list when adding nodes. The statement

```
Node p = new Node("Allan", myList);
```

adds a new node `"Allan"` to the beginning of the list of elements already accumulated. This statement uses the two-parameter constructor, which allows the successor of a node to be specified. The result of executing this sequence of four statements is the list of four elements shown at the bottom of Figure 19-1. The figure shows the `next` field of a `Node` object as an arrow pointing to a successor node. A backward diagonal drawn through the `next` field indicates a `null` value, and signifies that the node has no successor.

**Figure 19-1** Adding nodes to a linked list



```
myList = new Node("Bob");
myList.next = new Node("Carol");
myList.next.next = new Node("Debby");
Node p = new Node("Allan", myList);
```

In addition to adding nodes at the beginning and end of a list, we can insert nodes into the middle. Given the final scenario shown in Figure 19-1, we can add the string `"Brad"` immediately after `"Bob"` by writing

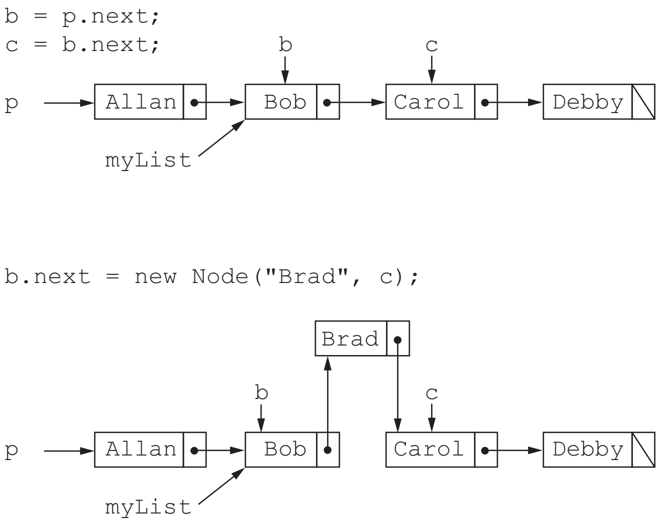
```
Node b = p.next;        // b points to "Bob"
Node c = b.next;        // c points to "Carol"
b.next = new Node("Brad", c);
```

Execution of these statements gives us the sequence of scenarios shown in Figure 19-2.

In the example just shown, we used two auxiliary references, `b` and `c`, to insert a node in the middle of a list. In general, adding a new node at any place other than the beginning requires a reference to the node that will be the predecessor (in this case, `b`) and to a node that will be the successor of the new node (in this case, `c`). The auxiliary variables are not always needed, however. For example, instead of using the three statements shown in Figure 19-2, we can add `"Brad"` to the list by writing

```
myList.next = new Node("Brad", myList.next);
```

**Figure 19-2** Inserting a node in the middle of a linked list



```
b = p.next;
c = b.next;
```

```
b.next = new Node("Brad", c);
```

b
c
p
Allan
Bob
Carol
Debby
myList
Brad
b
c
p
Allan
Bob
Carol
Debby
myList

## Removing a Node

You remove a node from a linked list by "routing" the successor link around it. Basically, to remove a node *X* (say `Carol` in the first list shown in Figure 19-3) from the middle of a linked list, you must first obtain references to its predecessor (In this case, `Bob`) and successor (In this case, `Debby`).

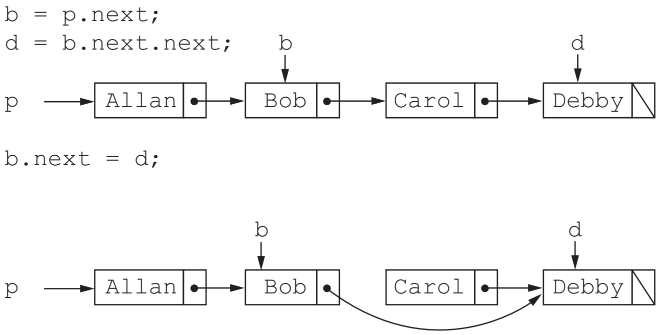
```
Node b = p.next;        // Predecessor of Carol
Node d = b.next.next;   // Successor of Carol
```

Then you remove the targeted node *X* by manipulating references to bypass *X* and make the successor of *X* the successor of its predecessor:

```
b.next = d;
```

This sequence of events is illustrated in Figure 19-3.

**Figure 19-3** Removing a node from the middle of a linked list



```
b = p.next;
d = b.next.next;
```

```
b.next = d;
```

b
d
p
Allan
Bob
Carol
Debby
b
d
p
Allan
Bob
Carol
Debby

As you may already have guessed, the removal of a node does not require an explicit reference to the successor of the targeted node. The following code fragment also removes `Carol` from the list, and is shorter:

```
b = p.next;
b.next = b.next.next;
```

The procedure we have just outlined for removing a node from the middle of a list will not work when the node being targeted for removal is the first in the list. This is because the first node has no predecessor. You remove a node at the beginning of a list by moving the head reference one node forward. For instance, to remove `"Allan"` from the list at the beginning of Figure 19-3, simply write:

```
p = p.next;
```

As we have seen, both addition and removal require a reference to the node just before the point of the addition or removal. You can obtain such a reference by setting a reference to the head of the list and then walking the reference forward, one node at a time, to get it to the desired place. The process of walking a reference along the nodes of a list is called traversing the list.

## Traversing Linked Lists

A *traversal* of a collection is a systematic method of doing some processing at each element of the collection. To traverse a linked list, set a reference, say `ref`, to the first node in the list, process the value at that node (by accessing `ref.value`) and then move on to the next node by setting `ref` to the successor of the current node:

```
ref = ref.next;
```

In this way, the reference `ref` moves from each node to the next until it becomes `null` at the end of the list. The following code fragment traverses a linked list `myList` and prints all of its elements:

```
Node ref = myList;
while (ref != null)
{
   System.out.println(ref.value);
   ref = ref.next;
}
```

Code Listing 19-1 builds a simple linked list and then traverses it to print its contents. The centerpiece of the program is the `LinkedList0` class. This class contains a private declaration of `Node`, which it uses to represent the linked list. `LinkedList0` uses a private field `first` to point to the first node of the linked list.

**Code Listing 19-1** (LinkedList0.java)

```
public class LinkedList0
{
   /**
      The Node class represents a list node.
   */

   private class Node
   {
      String value;
      Node next;

      /**
         Constructor.
         @param val The element to store in this node.
         @param n The reference to the next node.
      */

      Node(String val, Node n)
      {
         value = val;
         next = n;
      }

      /**
         Constructor.
         @param val The element to store in this node.
      */

      Node(String val)
      {
         value = val;
         next = null;
      }
   }

   //Reference to the first node in the list
   private Node first = null;

   /**
      Constructor.
      Builds a linked list.
   */

   public LinkedList0()
   {
      // Incrementally build the list
      // Chuck Debby Elaine Fred
      first = new Node("Debby");
      first.next = new Node("Elaine");
      first.next.next = new Node ("Fred");
      first = new Node ("Chuck", first);

      String [ ] names = {"Bob", "Allan"};

      // Use a loop to add all names in the array to the
      // front of the linked list to build the list
      // Allan Bob Chuck Debby Elaine Fred
      for (String s : names)
         first = new Node(s, first);
   }

   /**
      The print method traverses the list
      and prints all of its elements.
   */

   public void print()
   {
      Node ref = first;
      while (ref!= null)
      {
         System.out.print(ref.value + " ");
         ref = ref.next;
      }
   }

   /**
      The main method creates the linked list
      and invokes its print method.
   */

   public static void main(String [] args)
   {
      LinkedList0 ll = new LinkedList0();
      String str = "The contents of the list are:";
      System.out.println(str);
      ll.print();
   }
}
```

**Program Output**

```
The contents of the list are:
Allan Bob Chuck Debby Elaine Fred
```

**Checkpoint**

19.1 Give a sequence of statements that creates a linked list of strings `"red green blue"` by creating a node with `"green"`, then a node with `"blue"`, then a node with `"red"`, in that order.

19.2 Consider the following statements:

```
Node colorList = new Node("Yellow", new Node("Purple"));
colorList = new Node("Brown", colorList);
System.out.print(colorList.value);
System.out.print(colorList.next.value);
System.out.print(colorList.next.next.value);
```

What is the output?

## 19.2 Operations on Linked Lists

**CONCEPT: The most basic operations on linked lists are traversing the list, moving to a position determined by an index, adding an element, removing an element, and searching for an element.**

Let us write a linked list class that implements a subset of the methods of the `java.util.List` interface. Our class will implement the following methods:

- `boolean isEmpty()`: check if the list is empty.
- `int size()`: return the number of elements in the list.
- `void add(String e)`: add an element `e` to the end of the list.
- `void add(int index, String e)`: add the specified element `e` to the list at the position given by `index`.
- `String remove(int index)`: remove and return the element at the position given by `index`.
- `boolean remove(String e)`: remove the specified element `e` and return true if the item was successfully removed.

The `LinkedList1` class uses the same `Node` class as `LinkedList0`. The class also uses two references

```
Node first;
Node last;
```

to keep track of the first and last nodes in the list. The reference `first` serves as the head of the list, and `last` is used to enable elements to be added quickly to the end of the list. Both references will be set to `null` when the list is empty.

### The `isEmpty` Method

We can check to see if the list is empty by evaluating the expression `first == null`. The `isEmpty` method simply returns the value of that expression:

```
boolean isEmpty()
{
   return first == null;
}
```

## The `size` Method

The number of elements in a linked list can be determined by initializing a counter to 0 and then incrementing the counter at each step of a traversal of the linked list.

## The add Methods

The `add(String e)` method adds a new element `e` to the end of the linked list. If the list is empty, the new node containing `e` becomes both the first and last node:

```
first = new Node(e);
last = first;
```

If the list is nonempty, the new node is made the successor of the last node, and then the reference `last` is set to point to the new node:

```
last.next = new Node(e);
last = last.next;
```

The `add(int index, String e)` method is a little more involved. We consider several cases:

- *The index is invalid*. If the index is negative, or greater than the size of the list, the method throws an exception.
- *The index is zero*. The new node becomes the first node of the list. If the list was previously empty, then `last` must be set to `first`.
- *The index is greater than zero*. The method must find the right place to add the new node. It does this by skipping `index-1` nodes to get to the node that is to be the predecessor of the new node as follows:

  ```
  Node pred = first;
  for (int k = 1; k <= index - 1; k++)
  {
     pred = pred.next;
  }
  ```

Once the predecessor is found, the new node is created and spliced in as follows:

```
pred.next = new Node(e, pred.next);
```

At this point, the method checks to see if the new node is now the last node. If so, `last` must be updated to point to the new node as follows:

```
if (pred.next.next == null)
   last = pred.next;
```

## The `remove` Methods

The remove methods follow logic similar to the corresponding add methods. Let us consider the `remove(int index)` method. Again, we consider a number of different cases:

- *The index is invalid*. If the index is negative, or greater or equal to the size of the list, the method throws an exception.
- *The index is zero*. Here the element to be removed is the first on the list. The method moves `first` down to remove the first node. If the list is now empty, then `last` must be set to `null`.
- *The index is greater than zero*. In this case the method sets a reference `pred` to the predecessor of the node being targeted for removal. The target node is then removed. If the removed node was at the end of the list, then `last` must be reset to point to its predecessor:

```
pred.next == pred.next.next;
if (pred.next == null)
   last = pred;
```

## The `toString` Method

This method returns a string representation of the contents of the list, and is useful for displaying the list in a GUI component. The method appends the string value of each list element to a `StringBuilder` object and returns the string value representation of that object.

The `LinkedList1` class and all of its methods can be seen in Code Listing 19-2.

**Code Listing 19-2** (`LinkedList1.java`)

```
/**
   The LinkedList1 class implements a Linked list.
*/

class LinkedList1
{
   /**
      The Node class stores a list element
      and a reference to the next node.
   */

   private class Node
   {
      String value;
      Node next;

      /**
         Constructor.
         @param val The element to store in the node.
         @param n The reference to the successor node.
      */

      Node(String val, Node n)
      {
         value = val;
         next = n;
      }

      /**
         Constructor.
         @param val The element to store in the node.
      */

      Node(String val)
      {
         // Call the other (sister) constructor.
         this(val, null);
      }
   }

   private Node first;   // list head
   private Node last;    // last element in list

   /**
      Constructor.
   */

   public LinkedList1()
   {
      first = null;
      last = null;
   }

   /**
      The isEmpty method checks to see
      if the list is empty.
      @return true if list is empty,
      false otherwise.
   */

   public boolean isEmpty()
   {
      return first == null;
   }

   /**
```

```
      The size method returns the length of the list.
      @return The number of elements in the list.
   */

   public int size()
   {
      int count = 0;
      Node p = first;
      while (p != null)
      {
         // There is an element at p
         count ++;
         p = p.next;
      }
      return count;
   }

   /**
      The add method adds an element to
      the end of the list.
      @param e The value to add to the
      end of the list.
   */

   public void add(String e)
   {
      if (isEmpty())
      {
         first = new Node(e);
         last = first;
      }
      else
      {
         // Add to end of existing list
         last.next = new Node(e);
         last = last.next;
      }
   }

   /**
      The add method adds an element at a position.
      @param e The element to add to the list.
      @param index The position at which to add
      the element.
      @exception IndexOutOfBoundsException When
      index is out of bounds.
   */

```

```
    public void add(int index, String e)
    {
       if (index < 0  || index > size())
       {
          String message = String.valueOf(index);
          throw new IndexOutOfBoundsException(message);
       }

       // Index is at least 0
       if (index == 0)
       {
          // New element goes at beginning
          first = new Node(e, first);
          if (last == null)
             last = first;
          return;
       }

       // Set a reference pred to point to the node that
       // will be the predecessor of the new node
       Node pred = first;
       for (int k = 1; k <= index - 1; k++)
       {
          pred = pred.next;
       }

       // Splice in a node containing the new element
       pred.next = new Node(e, pred.next);

       // Is there a new last element ?
       if (pred.next.next == null)
          last = pred.next;
    }

    /**
       The toString method computes the string
       representation of the list.
       @return The string form of the list.
    */

    public String toString()
    {
       StringBuilder strBuilder = new StringBuilder();

       // Use p to walk down the linked list
       Node p = first;
       while (p != null)
       {
            strBuilder.append(p.value + "\n");
            p = p.next;
        }
        return strBuilder.toString();
    }

    /**
        The remove method removes the element at an index.
        @param index The index of the element to remove.
        @return The element removed.
        @exception IndexOutOfBoundsException When index is
                    out of bounds.
    */

    public String remove(int index)
    {
        if (index < 0 || index >= size())
        {
            String message = String.valueOf(index);
            throw new IndexOutOfBoundsException(message);
        }

        String element;  // The element to return
        if (index == 0)
        {
            // Removal of first item in the list
            element = first.value;
            first = first.next;
            if (first == null)
                last = null;
        }
        else
        {
            // To remove an element other than the first,
            // find the predecessor of the element to
            // be removed.
            Node pred = first;

            // Move pred forward index - 1 times
            for (int k = 1; k <= index -1; k++)
                pred = pred.next;

            // Store the value to return
            element = pred.next.value;

            // Route link around the node to be removed
            pred.next = pred.next.next;

```

```
         // Check if pred is now last
         if (pred.next == null)
            last = pred;
      }
      return element;
   }

   /**
      The remove method removes an element.
      @param element The element to remove.
      @return true if the remove succeeded,
      false otherwise.
   */

   public boolean remove(String element)
   {
      if (isEmpty())
         return false;

      if (element.equals(first.value))
      {
         // Removal of first item in the list
         first = first.next;
         if (first == null)
            last = null;
         return true;
      }

      // Find the predecessor of the element to remove
      Node pred = first;
      while (pred.next != null &&
            !pred.next.value.equals(element))
      {
         pred = pred.next;
      }

      // pred.next == null OR pred.next.value is element
      if (pred.next == null)
         return false;

      // pred.next.value  is element
      pred.next = pred.next.next;

      // Check if pred is now last
      if (pred.next == null)
         last = pred;

      return true;
```

```
    }

    public static void main(String [] args)
    {
        LinkedList1 ll = new LinkedList1();
        ll.add("Amy");
        ll.add("Bob");
        ll.add(0, "Al");
        ll.add(2, "Beth");
        ll.add(4, "Carol");
        System.out.println("The members of the list are:");
        System.out.print(ll);
    }
}
```

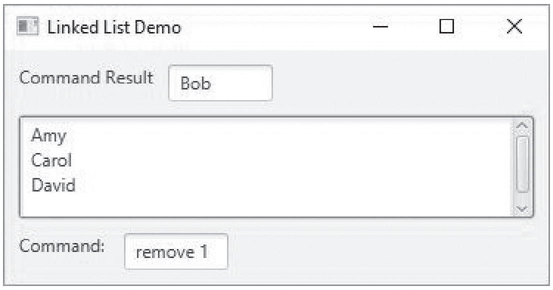
**Program Output**

```
The members of the list are:
Al
Amy
Beth
Bob
Caro
```

## A Graphical Interface for the Linked List Program

Now we describe a graphical user interface that can be used to test the `LinkedList1` class and other classes like it. The user interface is shown in Figure 19-4. The screenshot shows a scenario in which the user has given commands to add the strings `Amy`, `Bob`, `Carol`, and `David` to the list, followed by a command to remove the string at index 1 (`Bob`).

**Figure 19-4** A graphical interface for testing the linked list class



Command Result: Bob

Amy
Carol
David

Command: remove 1

The interface has a command entry text field at the bottom of the window, a text area in the center of the window, and an un-editable text field at the top. The text field at the top is used to display values returned by some of the list class methods.

The program that builds this user interface is shown in Code Listing 19-3. It attaches an event handler to the command text field at the bottom of the window, causing the handler to be called when the user types a command into the text field and presses the `Enter` key. The event handler method retrieves the typed command, analyzes it to determine its meaning, and calls the appropriate `LinkedList1` class method.

The program recognizes simple commands that are closely patterned after the methods they are intended to test. The list of recognized commands:

- `isempty`: This command calls the `isEmpty()` method and displays the returned value in the result text field at the top of the frame.
- `size`: This command calls `size()` and displays the returned value in the result text field.
- `add s`: This command calls the `add(String e)` method and passes it `s` in place of `e`. For example, the user might type `add Amy`.
- `add i s`: This command calls the `add(int index, string e)` method and passes it the argument `i` for `index` and `s` for `e`. As an example, the user might type `add 2 Brad` to add `"Brad"` as the third element in the list.
- `remove s`: This command calls the `remove(String e)` method and passes `s` for `e`.
- `remove i`: This command calls the `remove(int index)` method and passes `i` for `index`.

The listener method works by separating the text retrieved from the command entry text field into a command followed by its arguments. For example, the command text

```
add 2 Brad
```

is separated into three tokens: the command `add`, the integer index `2`, and the element `Brad`. The method creates a `Scanner` object that will read from the command string and then uses the usual `Scanner` methods to read strings and numbers from the command. This is done as follows:

```
String cmdText = cmdTextField.getText();
// Create a Scanner to read from the string
Scanner sc = new Scanner(cmdText);
```

The full details of the implementation can be found in Code Listing 19-3. The program uses a separate, non-nested class for the event handler. The class uses constructor parameters to pass the linked list object, the text area to be used for the list view, and the text field used to display results. The text field used for command entry is accessed through the `getTarget()` method of the event parameter of the event handler method.

**Code Listing 19-3** (`LinkedListDemo.java`)

```
import java.util.Scanner;
import javafx.application.Application;
import javafx.event.*;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

/**
    This class is used to demonstrate
    the operations in the LinkedList1 class.
*/

public class LinkedListDemo extends Application
{
    @Override
    public void start(Stage stage) throws Exception
    {
        // Reference to linked list.
        LinkedList1 ll = new LinkedList1();

        // listView shows current list elements.
        TextArea  listView = new TextArea();
        listView.setEditable(false);

        // Input for user command.
        TextField cmdTextField = new TextField();
        cmdTextField.setPrefColumnCount(5);

        // Displays result of list method that was
        // invoked by the user command.
        TextField resultTextField = new TextField();
        resultTextField.setPrefColumnCount(5);
        resultTextField.setEditable(false);

        // Attach event handler to cmdTextField
        EventHandler<ActionEvent> handler =
                new CommandHandler(ll, listView, resultTextField);
        cmdTextField.setOnAction(handler);

         // HBox to contain command result label and text field.
        HBox hBox1 = new HBox(10);
        Label resultLabel = new Label("Command Result");
        hBox1.getChildren().addAll(resultLabel, resultTextField);

        // HBox to contain label and text field for command input.
        HBox hBox2 = new HBox(10);
        Label cmdLabel = new Label("Command: ");
        hBox2.getChildren().addAll(cmdLabel, cmdTextField);

        // VBox to contain the user interface components
        VBox vBox = new VBox(10);
        vBox.setPadding(new Insets(10));
        vBox.getChildren().addAll(hBox1, listView, hBox2);

        // Set up the scene and show the stage.
        stage.setScene(new Scene(vBox));
        stage.setTitle("Linked List Demo");
        stage.show();
    }

    public static void main(String [] args)
    {
        launch();
    }
}

// Event Handler class for the command text field.
class CommandHandler implements EventHandler<ActionEvent>
{
    // Fields to hold information passed to the constructor.
    private final LinkedList1 ll;
    private final TextField resultTextField;
    private final TextArea listView;

    CommandHandler(LinkedList1 lList, TextArea lView,
                   TextField rTfield)
    {
        ll = lList;
        resultTextField = rTfield;
        listView = lView;
    }

    @Override
    public void handle(ActionEvent event)
    {
        // Get the command from the command textfield.
        TextField cmdTextField = (TextField)event.getTarget();
        String cmdText =  cmdTextField.getText();

        // Use a scanner to read the name of the linked list
        // method and do a switch on it.
        Scanner sc = new Scanner(cmdText);
        String cmd = sc.next();
        switch(cmd)
        {
            case "add":
                if (sc.hasNextInt())
                {
                    // add index element
                    int index = sc.nextInt();
                    String element = sc.next();
                    ll.add(index, element);
                }
                else
                {
                    // add element
                    String element = sc.next();
                    ll.add(element);
                }
                listView.setText(ll.toString());
                return;
            case "remove":
                if (sc.hasNextInt())
                {
                    // remove index
                    int index = sc.nextInt();
                    String res = ll.remove(index);
                    resultTextField.setText(res);
                }
                else
                {
                    // remove element
                    String element = sc.next();
                    boolean res = ll.remove(element);
                    String resText = String.valueOf(res);
                    resultTextField.setText(resText);
                }
                listView.setText(ll.toString());
                return;
            case "isempty":
                String resText = String.valueOf(ll.isEmpty());
                resultTextField.setText(resText);
                return;
            case "size":
               String resText1 = String.valueOf(ll.size());
               resultTextField.setText(resText1);
               return;
        }
    }
}
```

## 19.3 Doubly Linked and Circularly Linked Lists

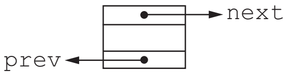
**CONCEPT: A singly linked list makes it easy to move from any node to its successor. A doubly linked list makes it easy to move from any node to its successor or predecessor. A circularly linked list makes it easy to move from one end of the list to the other.**

The singly linked lists we have been studying allow us to move from any node to its successor in one step. If we want to move from a node to its predecessor, though, we must start at the head of the list and then move all the way down to the predecessor of the current node. That gets costly if the current node is near the end of a long list. Moving to the predecessor can be done in one step if we add a predecessor reference to each node. Linked lists with successor and predecessor references are said to be *doubly linked*. Nodes in a doubly linked list look like this:

```
class Node
{
   String value; // Element
   Node next;    // Reference to successor
   Node prev;    // Reference to predecessor
   Node(String val, Node n, Node p)
   {
      value = val;
      next = n;
      prev = p;
   }

   Node(String val)
   {
      // Call the other constructor
      this(val, null, null);
   }
}
```

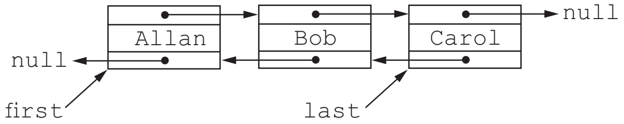
A node with `next` and `prev` references is shown in Figure 19-5.

**Figure 19-5** A node for a doubly linked list



Working with doubly linked lists is not much different from working with singly linked lists. You use a `null` reference to represent an empty list and a reference to the head of the list to represent a nonempty list. You set `next` to `null` if the node has no successor, and set `prev` to `null` if the node has no predecessor. An example of a doubly linked list with the names `Alan`, `Bob`, and `Carol` is shown in Figure 19-6. The list uses a head pointer named `first` and a pointer to the last element named `last`.

**Figure 19-6** A doubly linked list



## Adding a Node to a Doubly Linked List

VideoNote

Adding a Node to a Doubly Linked List

Adding a node to a doubly linked list is similar to adding a node to a singly linked one. Say we want to add an element `e` to a doubly linked list. If the list is empty, the new node will have neither successor nor predecessor, so we can write the following:

```
first = new Node(e, null, null);
last = first;
```

Alternatively, we can use the one-arg constructor as follows:

```
first = new Node(e);
last = first;
```

There are two cases to consider when the list is nonempty: when the new node is to be added *before* the first element, and when the new node is to be added *after*. To add the new node before the first element, we can write

```
first = new Node(e, first, null);
first.next.prev = first;
```

Now let us consider the latter case. One way to proceed is first to obtain a reference to the node that will become the predecessor of the new node. Let us say such a reference, `pred`, has already been obtained. Now take another reference, `succ`, and set it to the node following `pred`:

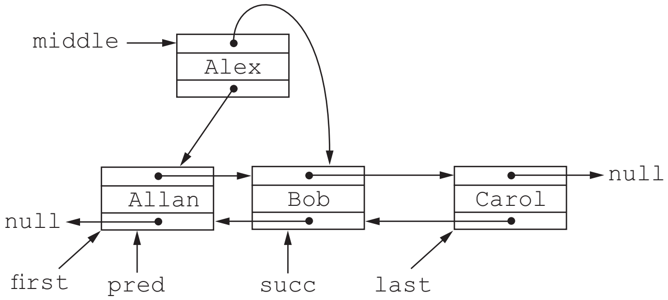
```
Node succ = pred.next;
```

The new node to be added will go between `pred` and `succ`. Create this new node using a reference named `middle`:

```
Node middle = new Node(e, succ, pred);
```

Executing this sequence results in the scenario shown in Figure 19-7, with `Alex` playing the role of the element `e`. The goal is to insert `Alex` right after `Allan`.

**Figure 19-7** The first step in inserting a node into a doubly linked list



Once the situation in Figure 19-7 is achieved, it only remains to set `pred.next` to `middle` and `succ.prev` to `middle`. But we must be careful: while we know that `pred` is not `null`, we cannot say the same about `succ`. Moreover, if `succ` is null, it means that the new node was added at the end of the list, and `last` must be set to point to it. The correct code to finish the job follows:

```
pred.next = middle;
if (succ == null)
   last = middle
else
   succ.prev = middle;
```

## Removing a Node from a Doubly Linked List

Removing a node from a doubly linked list is not difficult, although it needs to be done with some care. First, obtain a reference to the node you want to remove. Given such a reference, say `target`, obtain references to the nodes on either side:

```
Node pred = target.prev;
Node succ = target.next;
```

Now you want to make `succ` the successor of `pred`, and `pred` the predecessor of `succ`:

```
pred.next = succ;
succ.prev = pred;
```

Unfortunately, this code does not work if the targeted node is either the first or the last in the list. For example, if the targeted node is the last in the list, then `succ` will be `null` and trying to set `succ.prev` to `pred` will cause an exception to be thrown. The correct code follows:

```
if (pred == null)
   first = succ;
else
   pred.next = succ;

if (succ == null)
   last = pred;
else
   succ.prev = pred;
```

Note that this code works correctly even when the node being removed is the only one in the list. An implementation of a doubly linked list class is given in Code Listing 19-4.

**Code Listing 19-4** `(DLinkedList.java)`

```
/**
   The DLinkedList class implements a doubly
   Linked list.
*/

class DLinkedList
```

```
{
   /**
      The Node class stores a list element
      and a reference to the next node.
   */
   private class Node
   {
      String value;      // Value of a list element
      Node next;         // Next node in the list
      Node prev;         // Previous element in the list

       /**
          Constructor.
          @param val The element to be stored in the node.
          @param n The reference to the successor node.
          @param p The reference to the predecessor node.
       */

      Node(String val, Node n, Node p)
      {
         value = val;
         next = n;
         prev = p;
      }

      /**
         Constructor.
         @param val The element to be stored in the node.
      */

      Node(String val)
      {
         // Just call the other (sister) constructor
         this(val, null, null);
      }
   }

   private Node first;         // Head of the list
   private Node last;          // Last element on the list

   /**
      Constructor.
   */

   public DLinkedList()
   {
      first = null;
      last = null;
   }

   /**
      The isEmpty method checks to see if the list
      is empty.
      @return true if list is empty, false otherwise.
   */

   public boolean isEmpty()
   {
      return first == null;
   }

   /**
      The size method returns the length of the list.
      @return The number of elements in the list.
   */

   public int size()
   {
      int count = 0;
      Node p = first;
      while (p != null)
      {
         // There is an element at p
         count ++;
         p = p.next;
      }
      return count;
   }

   /**
      The add method adds to the end of the list.
      @param e The value to add.
   */

   public void add(String e)
   {
      if (isEmpty())
      {
         last = new Node(e);
         first = last;
      }
      else
      {
         // Add to end of existing list
         last.next = new Node(e, null, last);
         last = last.next;
      }
   }
```

```

    /**
       This add method adds an element at an index.
       @param e The element to add to the list.
       @param index The index at which to add.
       @exception IndexOutOfBoundsException
       When the index is out of bounds.
    */

    public void add(int index, String e)
    {
       if (index < 0  || index > size())
       {
          String message = String.valueOf(index);
          throw new IndexOutOfBoundsException(message);
       }

       // Index is at least 0
       if (index == 0)
       {
          // New element goes at beginning
          Node p = first;             // Old first
          first = new Node(e, p, null);
          if (p != null)
             p.prev = first;
          if (last == null)
             last = first;
          return;
       }

       // pred will point to the predecessor
       // of the new node.
       Node pred = first;
       for (int k = 1; k <= index - 1; k++)
       {
          pred = pred.next;
       }

       // Splice in a node with the new element
       // We want to go from  pred-- succ to
       // pred--middle--succ
       Node succ = pred.next;
       Node middle = new Node(e, succ, pred);
       pred.next = middle;
       if (succ == null)
          last = middle;
       else
          succ.prev = middle;
    }
```

```

   /**
      The toString method computes the string
      representation of the list.
      @return The string representation of the
      linked list.
   */

   public String toString()
   {
      StringBuilder strBuilder = new StringBuilder();

      // Use p to walk down the linked list
      Node p = first;
      while (p != null)
      {
         strBuilder.append(p.value + "\n");
         p = p.next;
      }
      return strBuilder.toString();
   }

   /**
      The remove method removes the element
      at a given position.
      @param index The position of the element
      to remove.
      @return The element removed.
      @exception IndexOutOfBoundsException When
      index is out of bounds.
   */

   public String remove(int index)
   {
      if (index < 0 || index >= size())
      {
         String message = String.valueOf(index);
         throw new IndexOutOfBoundsException(message);
      }

      // Locate the node targeted for removal
      Node target = first;
      for (int k = 1; k <= index; k++)
         target = target.next;

      String element = target.value;  // Element to return
      Node pred = target.prev;        // Node before the target
      Node succ = target.next;        // Node after the target

      // Route forward and back pointers around
      // the node to be removed
      if (pred == null)
         first = succ;
      else
         pred.next = succ;

      if (succ == null)
         last = pred;
      else
         succ.prev = pred;

      return element;
   }

   /**
      The remove method removes an element from the list.
      @param element The element to remove.
      @return true if the element was removed, false otherwise.
   */

   public boolean remove(String element)
   {
      if (isEmpty())
         return false;

      // Locate the node targeted for removal
      Node target = first;
      while (target != null
              && !element.equals(target.value))
         target = target.next;

      if (target == null)
         return false;

      Node pred = target.prev;        // Node before the target
      Node succ = target.next;        // Node after the target

      // Route forward and back pointers around
      // the node to be removed
      if (pred == null)
         first = succ;
      else
         pred.next = succ;

      if (succ == null)
         last = pred;
      else
         succ.prev = pred;

      return true;
   }

   public static void main(String [] args)
   {
      DLinkedList ll = new DLinkedList();
      ll.add("Amy");
      ll.add("Bob");
      ll.add(0, "Al");
      ll.add(2, "Beth");
      ll.add(4, "Carol");
      System.out.println("The elements of the list are:");
      System.out.println(ll);
   }
}
```

**Program Output**

```
The elements of the list are:
Al
Amy
Beth
Bob
Carol
```

This class can be used through a graphical user interface similar to that used with the `LinkedList1` class. The code that implements the interface is in the file *DLinkedListDemo.java*, and can be found in the book's source code, available at www.pearson.com/gaddis.

## Circularly Linked Lists

A singly linked list is *circularly linked* if the successor of the last node is the first. A doubly linked list is *circularly linked* if the successor of the last node is the first, and the predecessor of the first node is the last. Working with circularly linked lists is similar to working with non–circularly linked lists: the main differences are when you are adding to an empty list and when you are removing the last node from a list.

**Checkpoint**

19.3 Consider adding an element `e` to a nonempty doubly linked list with head pointer `first` and a pointer `last` pointing to the last node. The new node is to be added before some node that is already in the list. Let `succ` be a reference to the node in the list that will be the successor of the new node. Write code that can be used to add `e` to the list.

19.4 Consider adding an element `e` to a nonempty doubly linked list with head pointer `first` and a pointer `last` pointing to the last node. Write code that can be used to add `e` to the end of the list.

## 19.4 Recursion on Linked Lists

**CONCEPT: A linked list is an inherently recursive data structure, and many operations on linked lists have natural recursive implementations.**

The *tail* of a nonempty linked list *L* is the linked list that remains when the head of *L* is removed. The fact that the tail of a linked list is itself a linked list means that linked lists have a recursive structure. Many operations on linked lists can be given recursive formulations that process the element at the head of the list and then recursively operate on the tail. The base case for such recursive formulations is usually the case in which the list to be operated on is empty. This fact is very useful in formulating recursive methods that work on linked lists, so we restate it for emphasis:

*A linked list is either empty* (base case), *or consists of a head and a tail. The head is the first element of the list, and the tail is the linked list consisting of all the other elements.*

Consider a linked list class `RLinkedList` whose outline is shown here:

```
class RLinkedList
{
   private Node
   {

   }
   private Node first = null;
}
```

This class differs from the `LinkedList1` class of Section 19.2 in that it omits the reference that points to the last node in the list. We want to write recursive implementations of the `size`, `add`, and `remove` methods.

### A Recursive Implementation of `size`

The recursive strategy to compute the size of a list mirrors the recursive nature of the linked list: If the list is empty, its size is 0; otherwise, the size of the list is the size of the tail plus 1. Note that the tail of a list `list` is `list.next`. A recursive implementation of `size` follows:

```
private int size(Node list)
{
   if (list == null)
      return 0;                        // Base case
   else
      return size(list.next) + 1;      // Add 1 to size of tail
}
```

The public `size` method simply calls the private recursive version of `size`, passing it the reference to the head of the list as follows:

```
public int size( )
{
   return size(first);
}
```

Other recursive methods of this class will similarly be private, and will be called by a corresponding public method.

## A Recursive Implementation of the `add(String e, Node list)` Method

The `RLinkedList` class will have a recursive private method

```
Node add(String e, Node list)
```

that will be called by its public counterpart, void `add(String e)`. The recursive method will take a list `list` and an element `e` as parameters, and will return the list that results from adding `e` to the end of `list`. As in the case of `size`, the recursive strategy is based on the recursive nature of a linked list. If the list is empty, then the method returns a list of one item containing `e` as its only element. If the list is nonempty, then the element `e` is added to the end of the tail of the list as follows:

```
private Node add(String e, Node list)
{
   if (list == null)
   {
      // Base case
      return new Node(e);
   }
   else
   {
      // Add e to the end of the tail and use
      // the result to replace the tail
      list.next = add(e, list.next);
      return list;
   }
}
```

Notice the sequence of events in the nonbase case. First, the recursive call is used to return the result of adding `e` to the end of the tail:

```
add(e, list.next);
```

Second, this modified tail is used to replace the current tail in `list`:

```
list.next = add(e, list.next);
```

Third and finally, the method returns the original list with the tail thus replaced:

```
return list;
```

## A Recursive Implementation of `add(int index, String e, Node list)`

This method inserts an element `e` at a given index within a list and returns the resulting list. First the method checks that `index` is within the proper range and throws an exception if it is not. Suppose therefore that the index is nonnegative, and that the size of the list is greater than or equal to the index. Consider two cases:

- *The index is zero*. The method needs to put `e` at the beginning of the list (which may or may not be empty) and return the result. This is accomplished by

  ```
  return new Node(e, list);
  ```
- *The index is greater than zero*. Because the size of the list is greater than or equal to the index, the list cannot be empty, and therefore must have a head and a tail. The method needs to replace the tail with the result of adding `e` at position `index-1` in the tail and return the modified list:

  ```
  list.next = add(index-1, e, list.next);
  return list;
  ```

In this method, the recursion is on the index parameter rather than on the list, and the base case is when the index is zero. The complete method is as follows:

```
private Node add(int index, String e, Node list)
{
   if (index < 0  || index > size())
   {
      String message = String.valueOf(index);
      throw new IndexOutOfBoundsException(message);
   }

   if (index == 0)
      return new Node(e, list);

   list.next = add(index-1, e, list.next);
   return list;
}
```

## A Recursive Implementation of the `remove` Methods

Each of the two recursive remove methods returns a pair of results: the node removed from the list, and the list that remains after said node has been removed. These two results can be embodied in a single object of type `RemovalResult`:

```
private class RemovalResult
{
   Node node;  // The node removed from the list
   Node list;  // The list remaining after the removal

   RemovalResult(Node remNode, Node remList)
   {
      node = remNode;
      list = remList;
   }
}
```

First, consider the method

```
RemovalResult remove(int index, Node list)
```

This method is supposed to return the node at the given index, and the remaining list, packaged together in a single `RemovalResult` object. Assume that `index` is nonnegative and is less or equal to one less than the size of list. Again, consider the following cases:

- *The index is zero*. Then the size of the list must be greater or equal to 1, so the list is nonempty. The method needs to remove the first node from `list`, package the node and remaining list (which will be `list.next`), and return the resulting `RemovalResult` object.

  ```
  if (index == 0)
  {
     // Remove the first node on list
     RemovalResult remRes;
     remRes = new RemovalResult(list, list.next);
     list.next = null;
     return remRes;
  }
  ```

- *The index is nonzero*. In this case, the method needs to recursively remove the node at `index-1` from the tail:

  ```
  RemovalResult remRes;
  remRes = remove(index-1, list.next);
  ```

  The method then replaces the tail with the results of the recursive call:

  ```
  list.next = remRes.list;
  ```

  Before returning to its caller, the method must modify the `RemovalResult` object returned from the recursive call by setting its remaining list field to `list` (as modified by the replacement of its tail):

  ```
  remRes.list = list;
  return remRes;
  ```

The second removal method is similar and can be found in Code Listing 19-5.

**Code Listing 19-5** (RLinkedList.java)

```
/**
   Linked List class with recursive methods.
*/

class RLinkedList
{
   /**
      The Node class will store a list element
      and a reference to the next node.
   */

   private class Node
   {
      String value;
      Node next;

      /**
         Constructor.
         @param val The element to store in the node.
         @param n The reference to the successor node.
      */

      Node(String val, Node n)
      {
         value = val;
         next = n;
      }

      /**
         Constructor.
         @param val The element to be stored in the node.
       */

      Node(String val)
      {
         // Just call the other (sister) constructor
         this(val, null);
      }
   }

   private Node first;  // List head

   /**
      Constructor.
   */

   public RLinkedList()
   {
      first = null;
   }

   /**
      The isEmpty method checks to see if the list is empty.
      @return true if list is empty, false otherwise.
   */

   public boolean isEmpty()
   {
      return first == null;
   }

   /**
      The size method returns the length of the list.
      @return The number of elements in the list.
   */

   public int size()
   {
      return size(first);
   }

   /**
      This size method recursively computes the
      length of a linked list passed to it.
      @param The linked list.
      @return The number of items in the linked list.
   */

   private int size(Node list)
   {
      if (list == null)
         return 0;
      else
         return size(list.next) + 1;
   }

   /**
      The add method adds an element to the end of the list.
      @param e The value to add to the end of the list.
   */

   public void add(String e)
   {
      // Replace first with result of adding e to first
      first = add(e, first);
   }
```

```

    /**
       This recursive private add method adds
       an element e to the end of a list.
       @param e The element to add to the list.
       @param list The list to add e to.
       @return The list resulting from adding e to its end.
    */

    private Node add(String e, Node list)
    {
       if (list == null)
       {
          // Base case
          return new Node(e);
       }
       else
       {
          // Add e to the end of the tail and use
          // the result to replace the tail
          list.next = add(e, list.next);
          return list;
       }
    }

    /**
       The add method adds an element e at place index
       in this linked list.
       @param index The place in the list to add an element.
       @param e The element to add this the linked list.
       @exception IndexOutOfBoundsException When index is
                  out of bounds.
    */

    public void add(int index, String e)
    {
       // Replace first with the result of adding
       // e at index in first
       first = add(index, e, first);
    }

    /**
       This add method adds an element at an index in a list.
       @param e The element to add to the list.
       @param index The index at which to add the element.
       @param list The list to add e to.
       @return The list resulting from adding e.
       @exception IndexOutOfBoundsException When index is
                  out of bounds.
```

```
   */

   private Node add(int index, String e, Node list)
   {
      if (index < 0  || index > size())
      {
         String message = String.valueOf(index);
         throw new IndexOutOfBoundsException(message);
      }
      if (index == 0)
         return new Node(e, list);

      // 0 < index and index <= size so list is not empty
      // Replace the tail with result of adding e at index - 1
      // in the tail
      list.next = add(index-1, e, list.next);
      return list;
   }

   /**
      The toString method computes the string
      representation of the list.
      @return The string representation of the linked list.
   */

   public String toString()
   {
      StringBuilder strBuilder = new StringBuilder();

      // Use p to walk down the linked list
      Node p = first;
      while (p != null)
      {
         strBuilder.append(p.value + "\n");
         p = p.next;
      }
      return strBuilder.toString();
   }

   /**
      The RemovalResult class describes the results of
      removing a node from a linked list.
   */

   private class RemovalResult
   {
      Node node;   // The node removed from the list
      Node list;   // The list remaining after the removal
      RemovalResult(Node remNode, Node remList)
      {
         node = remNode;
         list = remList;
      }
   }

   /**
      The remove method removes the element at an index.
      @param index The index of the element to remove.
      @return The element removed.
      @exception IndexOutOfBoundsException When index is
                 out of bounds.
   */

   public String remove(int index)
   {
      // Pass the job on to the recursive version
      RemovalResult  remRes = remove(index, first);
      String element = remRes.node.value;  // Element to return
      first = remRes.list;                 // Remaining list
      return element;
   }

   /**
      The private remove method recursively removes
      the node at the given index from a list.
      @param index The position of the node to remove.
      @param list The list from which to remove a node.
      @return The result of removing the node from the list.
      @exception IndexOutOfBoundsException When index is
                 out of bounds.
   */
   private RemovalResult remove(int index, Node list)
   {
      if (index < 0 || index >= size())
      {
         String message = String.valueOf(index);
         throw new IndexOutOfBoundsException(message);
      }

      if (index == 0)
      {
         // Remove the first node on list
         RemovalResult remRes;
         remRes = new RemovalResult(list, list.next);
         list.next = null;
         return remRes;
      }

      // Recursively remove the element at index-1 in the tail
      RemovalResult remRes;
      remRes = remove(index-1, list.next);

      // Replace the tail with the results and return
      // after modifying the list part of RemovalResult
      list.next = remRes.list;
      remRes.list = list;
      return remRes;
   }

   /**
      The remove method removes a given
      element from linked list.
   */

   public boolean remove(String e)
   {
      RemovalResult remRes;
      remRes = remove(e, first);

      // Replace the list by the results of the removal
      first = remRes.list;
      if (remRes.node != null)
         return true;
      else
         return false;
   }

   /**
      The remove method recursively removes a
      node containing a given element from
      a specified list.
      @param element The element to remove.
      @param list The list to remove from.
      @return the list containing
   */

   private RemovalResult remove(String e, Node list)
   {
      if (list == null)
         return new RemovalResult(null, null);

      // Is the first node on list the target of the removal?
      if (list.value.equals(e))
      {
         RemovalResult remRes;
         remRes = new RemovalResult(list, list.next);
         list.next = null;
         return remRes;
      }

      // Node to be removed is after the first node on list
      // Recursively remove it from the tail
      RemovalResult remRes;
      remRes = remove(e, list.next);

      list.next = remRes.list;
      remRes.list = list;
      return remRes;
   }

   public static void main(String [] args)
   {
      RLinkedList ll = new RLinkedList();
      ll.add("Amy");
      ll.add("Bob");
      ll.add(0, "Al");
      ll.add(2, "Beth");
      ll.add(4, "Carol");
      System.out.println("The members of the list are:");
      System.out.print(ll);
   }
}
```

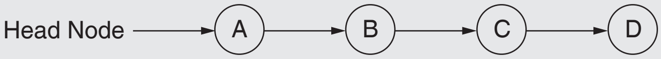
The book's source code, available at www.pearson.com/gaddis, contains a file *RLinkedListDemo.java* that implements a graphical user interface for this class.

## In the Spotlight:

### Directed Graphs

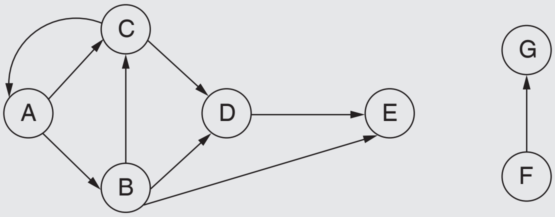
A linked list defines a one-way relation among a set of nodes in such a way that from any of the nodes you can move to at most one other node. If you can move from a node *A* to another node *B*, we say that *A* is *adjacent to B*, or that *B* is a *neighbor* of *A*. Using this terminology, we can see that a linked list is a collection of nodes in which every node may have at most one neighbor, and in which there is a path from a distinguished *head node* to every other node, as shown in Figure 19-8.

**Figure 19-8** The adjacency relation for nodes of a linked list



A directed graph is a data structure that is similar to a linked list, but with a much more general type of adjacency relation among its nodes. Specifically, a *directed graph* is a collection of nodes with an adjacency relation in which any node may have any number of neighbors. Directed graphs are often referred to as *digraphs*, and links between their nodes are called *edges*. An example of a directed graph is depicted in Figure 19-9.

**Figure 19-9** An example of a digraph



A digraph does not have to have a head node from which you can reach every other node, nor does it have to be "connected." Digraphs are very useful in modeling different types of systems. For example, a digraph can be used to model a computer network. In such a model, the nodes represent computers and the edges represent communication links.

The ideas we learned in this and preceding chapters can be used to create computer representations of digraphs. One such representation associates with each node *X* the list of its neighbors: that is, the list of nodes that can be reached from *X* by traversing a single edge. Such a list is called an *adjacency list* for *X*. For example, the digraph of Figure 19-9 can be represented as shown in Table 19-1.

**Table 19-1** Representing a digraph by adjacency lists

| Node | Neighbors |
|---|---|
| A | B, C |
| B | C, D, E |
| C | A, D |
| D | E |
| E | |
| F | G |
| G | |

The adjacency list representation can be realized by using a map that associates nodes with their adjacency lists. Code Listing 19-6 initializes an adjacency list for the digraph of Figure 19-9 and then prints it out. The program uses a `Map` of `String` to `List<String>`, which is defined in lines 12 and 13. In lines 22 through 27, the program acquires an entry set for the `Map` object and then uses the enhanced `for` loop to print out the adjacency list information. Entry sets for maps were discussed in Section 18.4.

**Code Listing 19-6** (AdjacencyList.java)

```java
import java.util.*;
/**
  This program demonstrates how to initialize an adjacency list
  structure whose nodes are strings. It uses a map of strings (nodes)
  to Lists of strings (nodes)
*/

public class AdjacencyList
{
    public static void main(String[] args)
    {
        Map<String, List<String>>
                          adjList = new HashMap<String, List<String>>();
        adjList.put("A", Arrays.asList("B", "C"));
        adjList.put("B", Arrays.asList("C", "D", "E"));
        adjList.put("C", Arrays.asList("A", "D"));
        adjList.put("D", Arrays.asList("E"));
        adjList.put("E", new ArrayList<String>());
        adjList.put("F", Arrays.asList("G"));
        adjList.put("G", new ArrayList<String>());

        Set< Map.Entry<String, List<String>>>
                                        entries = adjList.entrySet();
        for (Map.Entry<String, List<String>> e : entries)
        {
           System.out.println(e.getKey() + ": " + e.getValue()) ;
        }
    }
}
```

**Program Output**

```
D: [E]
E: []
F: [G]
G: []
A: [B, C]
B: [C, D, E]
C: [A, D]
```

In practice, we do not build adjacency lists of strings, as shown here. Instead, we establish a mapping between the set of names of the nodes and a set of nonnegative integers $0..N - 1$, where $N$ is the number of nodes. In our example, if we number the nodes $A$ through $G$ in order, the adjacency list structure might be as shown in Table 19-2.

**Table 19-2** An adjacency list using integers to represent digraph nodes

| Node | Neighbors |
|---|---|
| 0 | 1, 2 |
| 1 | 2, 3, 4 |
| 2 | 0, 3 |
| 3 | 4 |
| 4 | |
| 5 | 6 |
| 6 | |

The reason for the change is that, in practice, algorithms that work with adjacency lists often need to compare different nodes of the digraph for equality. It is more efficient to compare integers than it is to compare strings.

Data describing adjacency lists is usually stored in text files. Programs that work with digraphs read the data from such files and construct an in-memory representation of the adjacency list. The data stored in the file should follow a format that makes it easy for programs to *parse* the data. *Parsing* is the process of recognizing meaningful structure from syntax. In this case, we need to parse the file data to recognize which parts of the data represent nodes and which parts represent the adjacency lists for the nodes. Table 19-3 uses the example digraph of Figure 19-9 to illustrate a format that is relatively easy to parse.

**Table 19-3** Storing digraph information in a file to simplify parsing

```
7
A 0
B 1
C 2
D 3
E 4
F 5
G 6
0 2 1 2
1 3 2 3 4
2 2 0 3
3 1 4
4 0
5 1 6
6 0
```

The file starts with the number *N* of nodes in the graph, in this case 7. This number is followed by *N* lines, with each line showing the name of a node and the number that will be used to represent the node in the adjacency list. Following this is another set of *N* lines, with each line specifying the neighbors reachable from a single node. Each line has the form

(node) (number of reachable neighbors) (list of reachable neighbors)

For example, the line for node 1 is

```
1   3   2 3 4
```

This line shows that the node with number 1 has 3 reachable neighbors and that these neighbors form the list `2 3 4`.

## 19.5 Common Errors to Avoid

- **Forgetting to check that a reference to a node is not null.** Before you attempt to access the value stored in a node, or the `next` or `prev` references, make sure the reference is not null.
- **Forgetting to check for special cases.** Algorithms that operate on linked lists usually have to handle one or more special cases. Among the special cases to watch for are the following: an empty list when adding an element, a list of one element when removing an element, and the first or last node in the list when adding or removing a node.
- **Incorrect handling of base cases in recursive linked list algorithms.** Most recursive linked list algorithms do the recursion on the list parameter. In that case, the base case is usually when the list is empty. Algorithms that manipulate a list based on an index sometimes do the recursion on the index parameter: in that case the base case is when the index is zero.
- **Using the head of the list to traverse the list.** Do not use the head reference to walk down the list when you are traversing it, as it will destroy the structure of the list. Use an auxiliary reference initialized to the head.

## Review Questions and Exercises

### Multiple Choice and True/False

1. A list is a collection that __________.
   a. associates keys with elements
   b. assigns an index to each of its elements
   c. is implemented by the `JList` class
   d. none of the above
2. In Java, the first (head) element of a list __________.
   a. has a predecessor but no successor
   b. has both a predecessor and a successor
   c. has index 0
   d. has index 1

3. To add an element `e` just after a node referenced by `ref`, you should use the statement __________.
   a. `add(ref);`
   b. `ref.next = new Node(e);`
   c. `ref.next = new Node(e, ref.next);`
   d. `ref = new Node(e, ref.next);`
4. In general, adding an element `e` in *one step* to a singly linked list just before a node referenced by `ref` __________.
   a. cannot be done
   b. can be done if you use a loop
   c. can only be done when `ref` is the last node in the list
   d. can be done if you check for exceptions
5. To remove a node that is not the head of a singly linked list conveniently, you need to have a reference to __________.
   a. the node that precedes it
   b. the node that comes after it
   c. the node itself
   d. none of the above
6. To easily move from any node to its successor, and from any node to its predecessor, and from the first to the last node and from the last to the first node, you need __________.
   a. a singly linked list
   b. a singly and circularly linked list
   c. a doubly linked list
   d. a doubly and circularly linked list
7. A systematic method that starts at the beginning of the list and processes every node is called __________.
   a. a walk down the list
   b. a list processing function
   c. a travesty of list processing
   d. a traversal
8. A method for removing the element at an index *k* in a list throws an exception unless __________.
   a. *k* is negative
   b. *k* is nonnegative and less than the size of the list
   c. *k* is nonnegative and less than or equal to the size of the list
   d. *k* is positive and less than or equal to the size of the list
9. A method for adding an element at an index *k* in a list throws an exception unless __________.
   a. *k* is negative
   b. *k* is nonnegative and less than the size of the list
   c. *k* is nonnegative and less than or equal to the size of the list
   d. *k* is positive and less than or equal to the size of the list

10. A method that determines if a list is empty in a consistently written linked list class that uses a reference `first` to point to the first element, and a reference `last` to point to the last element, can return the value __________.
    a. `first == null`
    b. `last == null;`
    c. `first == null || last == null`
    d. all of the above
11. **True or False:** A linked list is a recursive data structure.
12. **True or False:** A linked list is a looping data structure.
13. **True or False:** The head of a linked list is also a linked list.
14. **True or False:** The tail of a linked list is also a linked list.
15. **True or False:** The index of an element to be added to a list can be the same as the size of the list.
16. **True or False:** The index of an element to be removed from a list can be the same as the size of the list.
17. **True or False:** The statement

    ```
    myList = new Node("Zoe", myList);
    ```

    adds `"Zoe"` to the end of the linked list `myList;`
18. **True or False:** There is a natural way to write a recursive method to check whether a linked list is empty.
19. **True or False:** The last item in a doubly linked list can sometimes have a successor.
20. **True or False:** The first item in a doubly linked list can sometimes have a successor.

### Find the Error

Find the error in each of the following code segments.

1. 
```
// Print the second element on
// a list myList of 3 elements
Node ref = myList;
ref ++ ;
System.out.print(ref.value);
```
2. 
```
// Print all elements in a list myList
Node ref = myList;
while (ref.next != null)
{
   System.out.print(ref.value + " ");
   ref = ref.next;
}
```
3. 
```
// Add a node to the beginning of
// a doubly linked list myList
myList = new Node("Abraham", myList, null);
```

4.
   ```
   // Remove the first node of a nonempty
   // doubly linked list myList
   myList = myList.next;
   ```

5.
   ```
   // The reference last points to the last
   // node in a nonempty doubly linked list.
   // Remove the last node from the list
   last = last.prev;
   ```

### Algorithm Workbench

1. Write a recursive method `void print(Node ref)` that prints the values of all elements in the linked list whose first node is `ref`.
2. Write a nonrecursive method `reverse()` that reverses the order of the elements in a list. Assume that this method is to be added to the `LinkedList1` class in this chapter.
3. Write a method `reverse()` as in Algorithm Workbench 2, except implement the method to call a private recursive method `Node reverse(Node list)`. Implement this recursive method as well.
4. Add a method `String removeMin()` to the `LinkedList1` class in this chapter. The method removes and returns the minimum string (according to the usual dictionary order on strings) from the list. If the list is empty, the method returns `null`.

### Short Answer

1. Explain why recursive linked list methods should not be made public.
2. Explain how to modify the linked list classes in this chapter so that the size of the list can be computed more efficiently. Describe specifically the changes that have to be made to the fields and methods of the class.
3. What is the index of an element in a list?
4. Explain how contiguous allocation works to store elements of a list.
5. Explain how linked allocation works to store elements of a list.
6. Consult the online Java documentation and determine the differences between the `StringBuffer` and `StringBuilder` classes.

## Programming Challenges

### 1. Generic Linked Lists

Modify the doubly linked list class presented in this chapter so it works with generic types. Add the following methods drawn from the `java.util.List` interface:

- `void clear()`: remove all elements from the list.
- `E get(int index)`: return the element at position `index` in the list.
- `E set(int index, E element)`: replace the element at the specified position with the specified element and return the previous element.

Test your generic linked list class by processing a list of numbers of type `double`.

### 2. Linked List Sorting and Reversing

Modify the `LinkedList1` class presented in this chapter by adding `sort()` and `reverse()` methods. The `reverse` method reverses the order of the elements in the list, and the `sort` method rearranges the elements in the list so they are sorted in alphabetical order. Do not use recursion to implement either of these operations. Extend the graphical interface in the `LinkedList1Demo` class to support sort and reverse commands, and use it to test the new methods.

VideoNote
Recursive Linked List Operations

### 3. Recursive Linked List Operations

Modify the `LinkedList1` class presented in this chapter by adding `sort()` and `reverse()` methods. The `reverse` method reverses the order of the elements in the list, and the `sort` method rearranges the elements in the list so they are sorted in alphabetical order. The class should use recursion to implement the sort and reverse operations. Extend the graphical interface in the `LinkedList1Demo` class to support sort and reverse commands, and use it to test the new methods.

### 4. Top Ten Gamers

Implement a class that maintains a list of the top 10 performers in a video game. An entry on the list consists of a name and score, and the list is kept sorted in descending order of scores. Here is an example of such a list when it has only four elements.

| | |
|---|---|
| Spike | 120 |
| Whiz | 105 |
| G-Man | 99 |
| JediMaster | 95 |

Use a class based on linked lists. The class should have a constructor that sets up an empty list, and a `void insert(String name, int score)` method that adds a name and a score pair to the list. The `insert` method puts the entry in the proper position so that the list stays sorted by score. The list should have a maximum size of 10. After the list has 10 elements, an attempt to add a name with a score that is less than or equal to the minimum score on the list is ignored, and adding a score that is greater than the minimum score causes an entry with the minimum score to be dropped from the list.

Test the score with a graphical user interface similar to `LinkedList1Demo.java`. The graphical interface should support a single command of the form

*insert name score*

An example of such a command is “insert Whiz 105.”

### 5. Wedding Lottery

A beautiful princess is to select a husband from $n$ suitors. The suitors are assigned numbers $0, \dots, n - 1$ and seated at a round table, and a random sequence of $n - 1$ numbers $x_0, x_1, \dots, x_{n-2}$, all of them in the range $0, \dots, n - 1$ is selected. The princess starts at suitor 0, and moving in a clockwise direction, she counts off $x_0$ positions and eliminates the suitor at that position. From there, she counts off another $x_1$ positions and eliminates the suitor at that position. She continues doing this util one lucky guy is left: she marries him and they live happily ever after.

Assume that the princess's position is always between two suitors. As an example, assume the suitors are *A*, *B*, *C*, and *D*, with *A* being numbered 0 and *D* being numbered 3. Let the random sequence be 0, 2, 1. Then the suitors will be eliminated in the order *A*, *D*, and *C*, and the lucky suitor will be *B*.

Write a class that uses linked lists to help the princess make her decision. The constructor for the class will be passed the number *n* of suitors (say 4), followed by a list of *n* suitors' names (say *A*, *B*, *C*, *D*), followed by a list of $n - 1$ numbers used to "rotate" among the suitors (0, 2, 1). The current position starts at the beginning of the list. The class should have a method `rotate(int x)` that counts off *x* steps in the clockwise direction beginning at the current position. The class should have a method

```
List<String> getEliminated()
```

that returns a list, in order of elimination, of the names of the suitors already eliminated. The class should also have a method

```
List <String> getHopeful()
```

that returns a list of suitors not yet eliminated.

Write a driver program that obtains the input *n*, a list of *n* suitors' names, and a list of $n - 1$ integers to be used as "rotate steps." Using the class you have written, the program prints the names of the rejected suitors (in the order of rejection), and then prints the name of the lucky guy.

### 6. Wedding Lottery II

Write an appropriate graphical user interface for the Wedding Lottery problem. The interface should allow the princess to specify the list of suitors using commands of the form *add name*. Once the princess has entered all the names, she presses a button labeled "Choose" to move from the input phase to the selection phase. In the selection phase, the user interface displays a list of suitors already rejected and another list of suitors still under consideration. The interface will have a text field in which the princess can enter a number for the next rotate step. As long as there is more than one suitor left, the program eliminates a suitor each time the princess types a number into the rotate step text field.

### 7. Adjacency List Input

Create a file containing the adjacency list data given at the end of this chapter, and then modify the program in Code Listing 19-6 so that it reads the file and uses the data found in the file to build the adjacency list structure. Use an array of strings to keep track of the association between node names and node numbers that is given at the beginning of the file, and then use an array or an `ArrayList` of lists of numbers for the adjacency list structure. Print the adjacency list twice: once using numbers to name the nodes (as in Table 19-3) and again using the string names of the nodes, as in Table 19-3.

CHAPTER

# 20 Stacks and Queues

**TOPICS**

20.1 Stacks and Their Applications
20.2 Array Implementation of Stacks
20.3 Linked Implementation of Stacks
20.4 Queues and Their Applications
20.5 Array Implementation of Queues
20.6 Linked List Implementation of Queues
20.7 Generic Implementation of Stacks and Queues
20.8 Queues and Breadth-First Search
20.9 Common Errors to Avoid

## 20.1 Stacks and Their Applications

**CONCEPT: A stack is a collection of items that allows addition and removal of items in a last-in-first-out manner.**

In Chapter 18 you learned about linked lists, array lists, and other types of collection classes that are part of the Java Collection Framework. The `List` classes you learned about in that chapter are very general, and allow elements to be added and removed at any position in the list. There are many applications that use lists of objects in more restricted ways. For example, an application may need to maintain a list of objects that requires additions and removals only at the ends, or even at only one end of the list. In a list that requires additions and removals to take place at only one end, the only item available for removal is the item that was last added. Such a list is called a *stack*, and the end of the list at which additions and removals take place is called the *top of the stack*.

### Examples and Applications of Stacks

There are many examples of stacks in real life. For example, when a number of cars are parked single file in a narrow driveway, the last car parked must be the first one to leave. Another example is a stack of plates in a cafeteria. Workers add plates to the top of the stack, and each patron removes a plate from the top of the stack.

There are many problems, both in computer science and out of it, that can be solved with the use of stacks. For example, all programming languages allow programmers to define methods and call them from different places within the program. The computer must keep track of the *return address* for each method call, that is, the place in the program from which the method was called. The computer then uses this return address when the method executes a return statement. Whenever a method is called, the computer adds its return address to a stack. Later, when the method executes a return, the computer removes an address from the stack and uses it as the target address for the return statement. The stack is the right data structure to use because the last method called is the first to return.

## Stack Operations

A stack can be viewed as an abstract data type that supports three main operations, traditionally called push, peek, and pop. The *push* operation takes a single item and adds it to the top of the stack. The *peek* operation returns the item currently at the top of the stack, but does not remove it. The *pop* operation removes (and returns) the item currently stored at the top of the stack.

The Java Collection Framework provides a `Stack` class that supports generic types. The class is in the `java.util` package. Table 20-1 lists most of its methods.

**Table 20-1** Stack class methods

| | |
|---|---|
| `Stack<E>()` | Constructor. |
| `E push(E item)` | Adds `item` to the top of the stack and returns the item added. |
| `E pop()` | Removes and returns the item at the top of the stack. If the stack is empty, `pop` throws `EmptyStackException`. |
| `E peek()` | Returns the item that is currently at the top of the stack, but does not remove it from the stack. If the stack is empty, `peek` throws `EmptyStackException`. |
| `boolean empty()` | Returns `true` if this stack is empty and `false` otherwise. |

Code Listing 20-1 is a simple demonstration of the use of the `Stack` class. The program creates a stack and pushes `String` objects onto it. Then it keeps popping the stack and printing the values popped until the stack is empty. Notice that the last-in-first-out nature of the stack causes the strings to be popped in the reverse of the order in which they were pushed.

**Code Listing 20-1** (`StackDemo1.java`)

```
import java.util.*;

/**
   This program demonstrates the java.util.Stack class.
*/

public class StackDemo1
{
   public static void main(String [] args)
   {
      // Create a stack of strings and add some names
      Stack<String> stack = new Stack<>();
      String [ ] names = {"Al", "Bob", "Carol"};
      System.out.println("Pushing onto the stack the names:");
      System.out.println("Al Bob Carol");
      for (String s : names)
         stack.push(s);

      // Now pop and print everything on the stack
      String message = "Popping and printing all stack values:";
      System.out.println(message);
      while (!stack.empty())
         System.out.print(stack.pop() + " ");
   }
}
```

**Program Output**

```
Pushing onto the stack the names:
Al Bob Carol
Popping and printing all stack values:
Carol Bob Al
```

## Stacks of Primitive Types

The `Stack` class provided by the Java Collections Framework does not directly accept values of primitive type. To use it with a primitive type such as `int` or `double`, you must use the corresponding wrapper class. For example, to have a stack of `int`, you must create a stack of `Integer`:

```
Stack<Integer> intStack = new Stack<>();
```

Then you can pass values of the appropriate primitive type to the `push` method, and the value will automatically be boxed. Likewise, popping a value from the stack and assigning to a variable of the appropriate primitive type will cause it to be unboxed. The use of stacks of primitive types is illustrated in the program of Code Listing 20-2. The program pushes some numbers onto a stack, then pops and prints them.

**Code Listing 20-2** `(StackDemo2.java)`

```
import java.util.*;

/**
   This program demonstrates the use
   of stacks with primitive types.
*/

```

```
public class StackDemo2
{
   public static void main(String [] args)
   {
      Stack<Integer> intStack = new Stack<>();

      // Push some numbers onto the stack
      for (int k = 1; k < 10; k++)
         intStack.push(k*k);

      // Pop and print all numbers
      while (!intStack.empty())
      {
         int x = intStack.pop();
         System.out.print( x + "  ");
      }
   }
}
```

**Program Output**

```
81  64  49  36  25  16  9  4  1
```

## 20.2 Array Implementation of Stacks

**CONCEPT: A stack can be implemented by using an array to hold the stack items and an integer to keep track of the top of the stack.**

A stack is just a list that enforces last-in-first-out access. In this section and the next, we look at how arrays and linked lists can be used to implement stacks.

The idea is simple. You create an array large enough to hold the largest number of items you will need to have in the stack at any given time, say, four (the capacity of a stack in a real application would likely be much larger), and you use an integer index, `top`, to point to the next slot in the array that is available to receive an item. Thus, for a stack of integers, you would have:

```
int [ ] s = new int [4];     // Array holds stack elements
int top = 0;                 // Pointer to top of stack
```

The variable `top`, when used in this manner, is traditionally called the *stack top pointer*, or simply, the *stack pointer*.

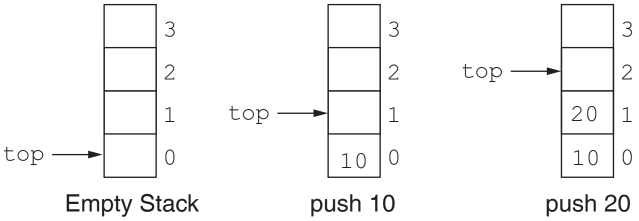
## The Stack Push Operation

To push a new item `x`, first we check to see if there is room in the stack. If so, we place it in the slot pointed to by `top` and then increment `top` to point to the next slot. If the stack has no room, we throw an exception as follows:

```
if (top == s.length)
   throw new StackOverFlowException();
else
{
   // Add the new item to the stack and update top
   s[top] = x;
   top ++;
}
```

`StackOverFlowException` is not a Java exception: it is one that we have to define. Figure 20-1 illustrates a sequence of events that starts with an empty stack and adds two items.

**Figure 20-1** Pushing items onto a stack



## The Stack `empty` Method

The boolean expression `top == 0` is true when the stack is empty and false when it is not, so the `empty` method simply returns the value of that expression:

```
return top == 0;
```

## The Stack `peek` and `pop` Methods

The two stack methods `peek` and `pop` are similar: both check to see if the stack is empty before attempting to access the item at the top of the stack, and throw `EmptyStackException` if the stack is empty. The `peek` method returns the item at the top of the stack as follows:

```
return s[top-1];
```

The `pop` method needs to return the same value as `peek`, but in addition, it needs to decrement the stack pointer to indicate that the stack slot currently holding the item at the top of the stack is becoming available. This is accomplished as follows:

```
top --;
return s[top];
```

The code we have outlined here will correctly implement a stack of integers. The stack class makes use of a separate file, *StackExceptions.java*, containing definitions of the following subclasses of the `RuntimeException` class. These two exception classes will be thrown by our stack methods.

```
class StackOverFlowException extends RuntimeException
{
}
class EmptyStackException extends RuntimeException
{
}
```

The code for the stack class is given next, as Code Listing 20-3.

**Code Listing 20-3** (`ArrayStack.java`)

```
/**
   Array Implementation of a stack.
*/

public class ArrayStack
{
   private int [] s; // Holds stack elements
   private int top;  // Stack top pointer

   /**
      Constructor.
      @param capacity The capacity of the stack.
   */

   public ArrayStack (int capacity)
   {
      s = new int[capacity];
      top = 0;
   }

   /**
      The empty method checks for an empty stack.
      @return true if stack is empty.
   */

   public boolean empty()
   {
      return top == 0;
   }

   /**
      The push method pushes a value onto the stack.
```

```
       @param x The value to push onto the stack.
       @exception StackOverflowException When the
       stack is full.
   */

   public void push(int x)
   {
      if (top == s.length)
         throw new StackOverFlowException();
      else
      {
         s[top] = x;
         top ++;
      }
   }

   /**
      The pop method pops a value off the stack.
      @return The value popped.
      @exception EmptyStackException When the
      stack is empty.
   */

   public int pop()
   {
      if (empty())
         throw new EmptyStackException();
      else
      {
         top--;
         return s[top];
      }
   }

   /**
      The peek method returns the value at the
      top of the stack.
      @return value at top of the stack.
      @exception EmptyStackException When the
      stack is empty.
   */

   int peek()
   {
      if (empty())
         throw new EmptyStackException();
      else
       {
            return s[top-1];
        }
    }
}
```

The `ArrayStackDemo` class, shown in Code Listing 20-4, demonstrates the use of this stack. It has a main method that creates a stack with capacity 5, stores some integers on it, and then pops and prints them.

**Code Listing 20-4** `(ArrayStackDemo.java)`

```
/**
   This class demonstrates the
   use of the ArrayStack class.
*/

public class ArrayStackDemo
{
   public static void main(String [] arg)
   {
      String str;  // Use for output
      ArrayStack  st = new ArrayStack(5);
      str = "Pushing 10 20 onto the stack.";
      System.out.println(str);
      st.push(10);
      st.push(20);
      str = "Value at top of the stack is ";
      System.out.println(str + st.peek());
      str = "Popping and printing all values:";
      System.out.println(str);
      while (!st.empty())
         System.out.print(st.pop() + " ");
   }
}
```

**Program Output**

```
Pushing 10 20 onto the stack.
Value at top of the stack is 20
Popping and printing all values:
20 10
```

## Stacks of Objects

The stack in Code Listing 20-3 is a stack of integers. Writing a stack that works with other types is similar. For example, to write a stack that works with `String` objects, we would modify the `ArrayStack` class by changing all occurrences of `int` that refer to the type of the items being stored in the stack to `String`. In particular, the parameter and return types of `peek`, `pop`, and `push` would change as follows:

```
int peek()          becomes    String peek()
int pop()           becomes    String pop()
void push(int x)    becomes    void push(String x)
```

The type of array that holds the stack items also needs to change as follows:

```
int [ ] s;                  becomes    String [ ] s;
s = new int[capacity]       becomes    s = new String[capacity];
```

One other change must be made to the `pop` method when the stack is being used to store object types: the array entry referring to the stack item being removed must be set to `null` to facilitate garbage collection of the item being removed from the stack as follows:

```
s[top-1] = null;
```

For example, the `pop` method modified to work with string objects would look like this:

```
public String pop()
{
   if (empty())
      throw new EmptyStackException();
   else
   {
      String retVal = s[top-1];
      s[top-1] = null; // Facilitate garbage collection
      top--;
      return retVal;
   }
}
```

Later in this chapter we will look at how we can use generic types to write a single stack class that works with values of all types.

A disadvantage of using an array implementation is the need to fix the size of the array at the time the stack is created. This causes a problem if the application using the stack needs to store more items than the size of the stack will allow. Basing the implementation of an array on a linked list avoids this problem.

## 20.3 Linked Implementation of Stacks

**CONCEPT: A stack can be implemented by using a singly linked list to hold the stack items, and having the head of the list serve as the top of the stack.**

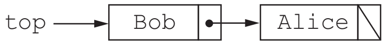
A stack can be implemented by using any type of list object, including the `LinkedList` class in the `java.util` package, to hold the stack items. The `LinkedList` class, however, supports many operations that are not needed when all we need is a stack: for example, most applications of stacks do not need iterators, or the ability to add and remove items from the middle of the list. We can avoid this additional overhead by basing a stack on a simple linked list that we code ourselves.

The linked list will be based on the following `Node` class, declared private inside a `LinkedStack` class:

```
private class Node
{
   String value;
   Node next;
   Node(String val, Node n)
   {
      value = val;
      next = n;
   }
}
```

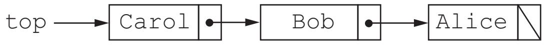
Inside the `LinkedStack` class, we will have a reference to a linked list of items. A reference `top` will point to the node at the front of the list: this node will be the one removed by the next call to `pop`. For example, starting with an empty stack and pushing the strings `"Alice"` and `"Bob"` in succession will result in the scenario shown in Figure 20-2.

**Figure 20-2** A linked stack after pushing two items



A further operation of pushing `"Carol"` on the stack would give the result shown in Figure 20-3:

**Figure 20-3** The linked stack of Figure 20-2 after push of the string `"Carol"`



At this point, executing a pop operation would remove `"Carol"` from the stack and return us to the scenario shown in Figure 20-2.

## Implementation of Stack Methods

The `LinkedStack` class will have a field `top` that points to the item at the top of the stack and simultaneously serves as the head of the linked list. The stack will be empty whenever the expression

```
top == null
```

is true, so the `empty` method simply returns the value of that expression:

```
public boolean empty()
{
   return top == null;
}
```

The `push` operation adds a new item `s` to the front of the list:

```
top = new Node(s, top);
```

The `peek` method returns the value field of the node at the front of the list:

```
return top.value;
```

The `pop` method saves the value in the node at the top, removes the top node from the list by moving the head pointer past it, and returns the saved value:

```
String retValue = top.value;
top = top.next;
return retValue;
```

Both `peek` and `pop` throw an exception if they are called on an empty stack. Finally, the `LinkedStack` class is equipped with a `toString` method that uses a `StringBuilder` object to create a string representation of the class. Having this method allows the contents of the entire stack `st` to be displayed by passing it to the `System.out.print` method:

```
System.out.println(st);
```

The `LinkedStack` class can be seen in Code Listing 20-5.

**Code Listing 20-5** (`LinkedStack.java`)

```
/**
   This program demonstrates how to write and
   use a stack class based on linked lists.
*/

class LinkedStack
{
   /**
      The Node class is used to implement the
      linked list.
   */

   private class Node
   {
      String value;
      Node next;
      Node(String val, Node n)
      {
         value = val;
         next = n;
      }
   }

   private Node top = null;  // Top of the stack

   /**
      The empty method checks for an empty stack.
      @return true if stack is empty, false otherwise.
   */

   public boolean empty()
   {
      return top == null;
   }

   /**
      The push method adds a new item to the stack.
      @param s The item to be pushed onto the stack.
   */

   public void push(String s)
   {
      top = new Node(s, top);
   }

   /**
      The Pop method removes the value at the
      top of the stack.
      @return The value at the top of the stack.
      @exception EmptyStackException When the
      stack is empty.
   */

   public String pop()
   {
      if (empty())
         throw new EmptyStackException();
      else
      {
         String retValue = top.value;
         top = top.next;
         return retValue;
      }
   }

   /**
      The peek method returns the top value
      on the stack.
      @return The value at the top of the stack.
      @exception EmptyStackException When the
      stack is empty.
   */

   public String peek()
   {
      if (empty())
         throw new EmptyStackException();
      else
         return top.value;
   }

   /**
      The toString method computes a string
      representation of the contents of the stack.
      @return The string representation of the
      stack contents.
   */

   public String toString()
   {
      StringBuilder sBuilder = new StringBuilder();
      Node p = top;
      while (p != null)
      {
         sBuilder.append(p.value);
         p = p.next;
         if (p != null)
            sBuilder.append("\n");
      }
      return sBuilder.toString();
   }
}
```

Code Listing 20-6 gives the `LinkedStackDemo` class, which demonstrates the use of the `LinkedStack` class.

**Code Listing 20-6** (LinkedStackDemo.java)

```java
/**
   This class demonstrates the use of the
   LinkedStack class.
*/

public class LinkedStackDemo
{
   public static void main(String [ ] args)
   {
      LinkedStack st = new LinkedStack();
      System.out.println("Pushing: Amy Bob Chuck");
      System.out.println("Contents of Stack:");
      st.push("Amy");
      st.push("Bob");
      st.push("Chuck");
      System.out.println(st);
      String name = st.pop();
      System.out.println("Popped: " + name);
      System.out.println("Contents of Stack:");
      System.out.println(st);
   }
}
```

**Program Output**

```
Pushing: Amy Bob Chuck
Contents of Stack:
Chuck
Bob
Amy
Popped: Chuck
Contents of Stack:
Bob
Amy
```

### Checkpoint

20.1 What is the common name for an operation that adds an element to a stack?

20.2 What is the common name for an operation that removes an element from a stack?

20.3 What should a stack method for removing an item do when the stack is empty?

20.4 In what order are elements added and removed from a stack?

20.5 Cite an example from everyday life where a collection of items behaves like a stack.

## 20.4 Queues and Their Applications

**CONCEPT: A queue is a collection of items that is accessed in a first-in-first-out fashion.**

You saw in the last section that a stack is a collection of items that is accessed in a last-in-first-out fashion. A queue is like a stack, except its items are added and removed in a first-in-first-out fashion. Queues, like stacks, have many applications in the real world. For example, at a grocery store checkout line, the first customer in the line is the first to be served.

Queues find many applications in computer science. Consider, for example, a network of computers that must share one printer. The printer server associates a queue with the printer, and print jobs that arrive from the various computers are added to this queue. Print jobs are removed from this queue and are serviced in a first-come-first-served fashion.

A queue can be viewed as a list where one end is designated as the *front* of the queue, and the other end is designated as the *rear*. We add items at the rear of the queue and remove items from the front. A queue is required to support the following operations:

- *enqueue*(*x*): add a new item *x* to the rear of the queue
- *dequeue*( ): remove and return the item at the front of the queue
- *empty*( ): check if the queue is empty
- *peek*( ): return, but do not remove, the item at the front of the queue

## 20.5 Array Implementation of Queues

**CONCEPT: Fixed size arrays can be used to implement queues.**

VideoNote
Array Implementation of Queues

To implement a queue using an array, we need an array `q`, and two integer indices `front` and `rear`. Assuming we want to implement a queue that will hold strings, we declare these variables as follows:

```
String [ ] q;
int front, rear;
```

It is useful to establish a condition that specifies how the variables `front` and `rear` are being used, and require that all queue methods that manipulate `front` and `rear` preserve this condition. This condition, which we will refer to as the *queue invariant*, is as follows:

- `front` is the index of the slot in `q` that holds the next item that will be dequeued. This slot will normally be *filled*.
- `rear` is the index of the slot in `q` that will hold the next item that will be enqueued. This slot will normally be *empty*.

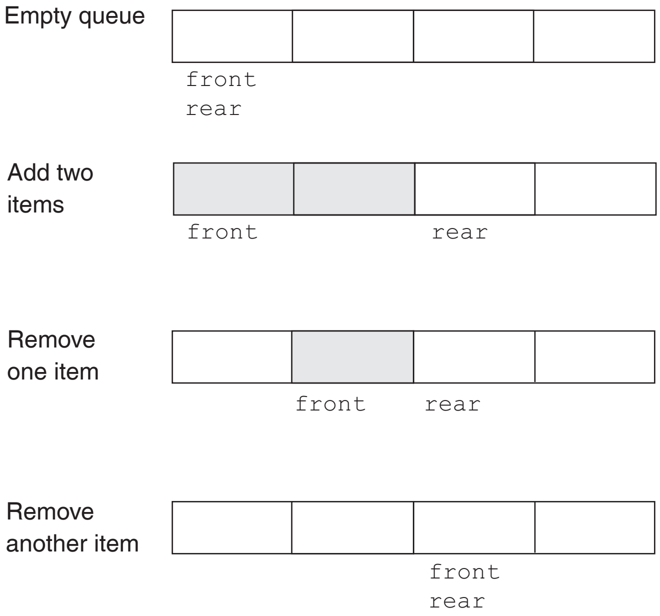
Initially, the queue is empty, and both `front` and `rear` are initialized to 0 as required by the queue invariant (the first element stored will be placed at 0, and the first element removed will come from 0). The queue invariant is illustrated in Figure 20-4.

Now suppose we want to add an item `s` to the queue. The queue invariant says that we should store `s` in the slot `q[rear]` and then increment `rear` so it points to the next empty slot as follows:

```
q[rear] = s;
rear ++;
```

Thus, after adding one item, `rear` will be 1, and after adding a second, `rear` will be 2. At this point `front` will still be 0. This is illustrated in the first two rows of Figure 20-4.

**Figure 20-4** Use of an array-based queue



Now consider the operation of removing an item. The item to be removed is `q[front]`. We save the value at that position so it can be returned to the caller, set the queue entry to `null`, and increment `front` so that it points to the next item that will be dequeued.

```
String value = q[front];
q[front] = null;
front ++;
```

This is illustrated in the third and fourth rows of Figure 20-4. Notice that as long as we have not reached the end of the array, `rear` will increase as items are added, and `front` will increase as items are removed, and `rear` and `front` will be equal precisely when the queue is empty.

Now suppose we start with the queue shown in the last row of Figure 20-4 and add two more items. At that point `front` will still be 2, and `rear` will be 0 because that is the next available empty slot in the array. That situation is shown in the top row of Figure 20-5. We see that to make the best use of available space in the array, the indices `front` and `rear` need to wrap around to 0 whenever they get to the end of the array. Accordingly, the correct code for adding an item to the queue is

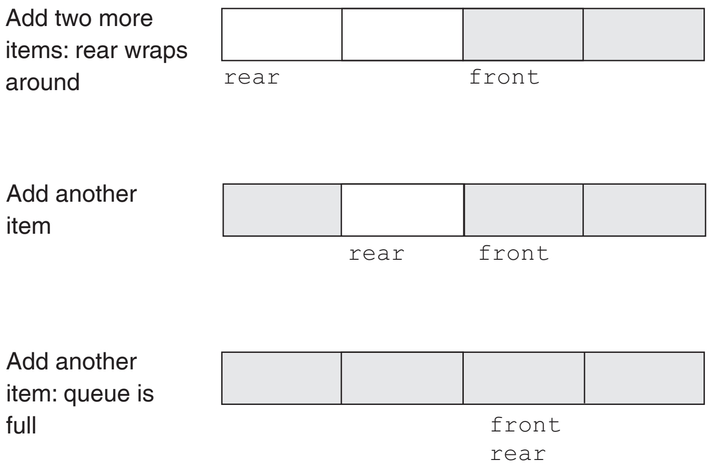
```
q[rear]= s;
rear ++;
if (rear == q.length) rear = 0;
```

and the correct code for removing an item is

```
String value = q[front];
q[front] = null;
front ++;
if (front == q.length) front = 0;
```

When array indices wrap around to the beginning of the array in this fashion, we say that we are using the array as a *circular buffer*.

**Figure 20-5** Use of an array as a circular buffer


Add two more items: rear wraps around
rear
front
Add another item
rear
front
Add another item: queue is full
front
rear

Notice in Figure 20-5 that `front` is also equal to `rear` when the array is completely full. This makes it impossible to distinguish a queue that is full from one that is empty just by looking at `front` and `rear`. One way to work around this is to have the queue keep a count of the number of items it contains: an enqueue operation increments this counter, and a dequeue operation decrements it. Then we can tell that the queue is empty when this count is 0, and that the queue is full when the count is equal to the length of the array.

A complete implementation of a queue is given in Code Listing 20-7. It uses a variable named `size` to keep track of the number of items in the queue, and has a method named `capacity` that returns the length of the array used inside the queue. Notice that the `enqueue` and `dequeue` methods check for an empty or full queue, and throw appropriately named exceptions when they cannot accomplish their designated task. The exceptions thrown are in a separate file named *QueueExceptions.java*:

```
class QueueOverFlowException extends RuntimeException
{
}

class EmptyQueueException extends RuntimeException
{
}
```

The `ArrayQueue` class in Code Listing 20-7 overrides the `toString` method to return a readable string representation of the contents of the queue. The string returned from this method encodes the current value of the `front` and `rear` indices, together with the value and position of each item in the queue. You can see the format of the string returned by the `toString` method, and how it encodes the state of the queue, in the Program Output following Code Listing 20-8.

**Code Listing 20-7** **(ArrayQueue.java)**

```
import java.util.*;

/**
   The ArrayQueue class uses an
   array to implement a queue.
*/

class ArrayQueue
{
   private String [ ] q;  // Holds queue elements
   private int front;     // Next item to be removed
   private int rear;      // Next slot to fill
   private int size;      // Number of items in queue

   /**
      Constructor.
      @param capacity  The capacity of the queue.
   */

   ArrayQueue(int capacity)
   {
      q = new String[capacity];
      front = 0;
      rear = 0;
      size = 0;
   }

   /**
      The capacity method returns the length of
      the array used to implement the queue.
      @return The capacity of the queue.
   */

   public int capacity()
   {
      return q.length;
   }

   /**
      The enqueue method adds an element to the queue.
      @param s The element to be added to the queue.
      @exception QueueOverFlowException When there
      is no more room in the queue.
   */

   public void enqueue(String s)
   {
      if (size == q.length)
         throw new QueueOverFlowException();
      else
      {
         // Add to rear
         size ++;
         q[rear] = s;
         rear ++;
         if (rear == q.length) rear = 0;
      }
   }

   /**
      The peek method returns the item
      at the front of the queue.
      @return element at front of queue.
      @exception EmptyQueueException When
      the queue is empty.
   */

   public String peek()
   {
      if (empty())
         throw new EmptyQueueException();
      else
         return q[front];
   }

   /**
      The dequeue method removes and returns
      the element at the front of the queue.
      @return The element at the front of the queue.
      @exception EmptyQueueException When
      the queue is empty.
   */

   public String dequeue()
   {
      if (empty())
         throw new EmptyQueueException();
      else
       {
         size--;
         // Remove from front
         String value = q[front];

         // Facilitate garbage collection
         q[front] = null;

         // Update front
         front++;
         if (front == q.length) front = 0;

         return value;
      }
   }

   /**
      The empty method checks to see if
      this queue is empty.
      @return true if the queue is empty and
      false otherwise.
   */

   public boolean empty()
   {
      return size == 0;
   }

   /**
      The toString method returns a
      readable representation of the
      contents of the queue.
      @return  The string representation
      of the contents of the queue.
   */

   public String toString()
   {
      StringBuilder sBuilder = new StringBuilder();
      sBuilder.append("front = " + front + "; ");
      sBuilder.append("rear = " + rear + "\n");
      for (int k = 0; k < q.length; k++)
      {
         if (q[k] != null)
            sBuilder.append(k + " " + q[k]);
         else
            sBuilder.append(k + " ?");
         if (k != q.length - 1)
            sBuilder.append("\n");
      }
      return sBuilder.toString();
   }
}
```

The program shown in Code Listing 20-8 demonstrates the use of the `ArrayQueue` class.

**Code Listing 20-8** (`ArrayQueueDemo.java`)

```
/**
   The ArrayQueueDemo demonstrates the use
   of the ArrayQueue class.
*/

public class ArrayQueueDemo
{
   public static void main(String [] args)
   {
      String str;  // Holds various string values

      ArrayQueue queue = new ArrayQueue(4);
      str = "Queue has capacity ";
      System.out.println(str + queue.capacity());

      // Add 4 names
      String [ ] names
         = {"Alfonso", "Bob", "Carol", "Deborah"};
      System.out.println("Adding names: ");
      for (String s : names)
      {
         System.out.print(s + " ");
         queue.enqueue(s);
      }
      System.out.println("\nState of queue is: ");
      System.out.println(queue);

```

```
        // Remove 2 names
        System.out.println("Removing 2 names.");
        queue.dequeue(); queue.dequeue();
        System.out.println("State of queue is: ");
        System.out.println(queue);

        // Add a name
        System.out.println("Adding the name Elaine:");
        queue.enqueue("Elaine");
        System.out.println("State of queue is: ");
        System.out.println(queue);
    }
}
```

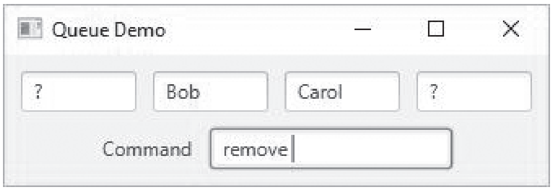
**Program Output**

```
Queue has capacity 4
Adding names:
Alfonso Bob Carol Deborah
State of queue is:
front = 0; rear = 0
0 Alfonso
1 Bob
2 Carol
3 Deborah
Removing 2 names.
State of queue is:
front = 2; rear = 0
0 ?
1 ?
2 Carol
3 Deborah
Adding the name Elaine:
State of queue is:
front = 2; rear = 1
0 Elaine
1 ?
2 Carol
3 Deborah
```

## GUI Front End for the Queue Demo Program

Now we describe a graphical user interface to the queue implemented in Code Listing 20-7. A screenshot of the user interface after the user starts with a queue of size four, adds three names, and then removes a name, is shown in Figure 20-6.

**Figure 20-6** Screenshot of GUI front end for the queue program (Mark Schrier. Copyright Oracle, Inc.)



? | Bob | Carol | ?

Command: remove

The user interface consists of a window with a pane at the top that has an array of text fields. At the bottom of the window, there is a text field used by the user to enter commands. The program recognizes the following commands:

- `enqueue x`: add x to the queue
- `add x`: add x to the queue (same as `enqueue x`)
- `dequeue`: remove an item from the queue
- `remove`: remove an item from the queue (same as `dequeue`)

The program sets an event handler on the command entry text field that is called whenever the user types a command. The handler retrieves the text of the command, parses it by using a `Scanner` object, identifies the command and its argument, and calls the appropriate queue method. The handler then calls the `refresh` method, passing it a string that represents the current state of the queue. The `refresh` method examines its argument to determine the position and value of each queue element, and then updates the queue view text fields at the top of the frame. Code Listing 20-9 gives the rest of the details.

**Code Listing 20-9** (`QueueDemo.java`)

```
import java.util.Scanner;
import javafx.application.Application;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

/**
    This program demonstrates the graphical display of a queue.
*/

public class QueueDemo extends Application
{
    public static void main(String [] args)
    {
       launch();
    }

    @Override
    public void start(Stage stage) throws Exception
    {
        // Create a queue.
        ArrayQueue queue = new ArrayQueue(4);

        // Create the GUI to display the queue.
        TextField[] qViewTextField = new TextField[4];
        TextField commandEntryTextField = new TextField();

        // Put the queue view text fields in a horizontal box.
        HBox topHBox = new HBox(10);

        for (int k = 0; k < qViewTextField.length; k++)
        {
            TextField tF = new TextField();
            tF.setPrefColumnCount(5);
            tF.setEditable(false);
            topHBox.getChildren().add(tF);
            qViewTextField[k] = tF;
        }

        // Put the label and commandEntryTextField in
        // a horizontal box.
        HBox hBox = new HBox(10);
        hBox.setAlignment(Pos.CENTER);
        hBox.getChildren().addAll(new Label("Command"),
                                  commandEntryTextField);

        // Use an outer VBox to hold all the GUI components
        VBox outerPane = new VBox(10, topHBox, hBox);
        outerPane.setPadding(new Insets(10));

        // Set the event handler on the text field
        EventHandler<ActionEvent> handler
                = new CmdTextHandler(queue, qViewTextField);
        commandEntryTextField.setOnAction(handler);

        stage.setScene(new Scene(outerPane));
        stage.setTitle("Queue Demo");
        stage.show();
    }
}

/**
   The handler for the command entry text field uses
   constructor parameters to store reference to
   the queue and the queue view text fields.
*/
```

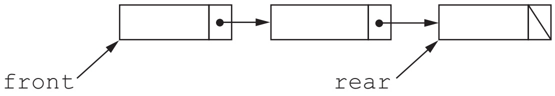
```
class CmdTextHandler implements EventHandler<ActionEvent>
{
    private ArrayQueue queue;
    private TextField[] qViewTextField;

    public CmdTextHandler(ArrayQueue queue1, TextField[] qViewTfs)
    {
        queue = queue1;
        qViewTextField = qViewTfs;
    }

    public void handle(ActionEvent evt)
    {
        TextField cmdTextField = (TextField) evt.getTarget();
        String cmdText = cmdTextField.getText();
        Scanner sc = new Scanner(cmdText);
        if (!sc.hasNext())
        {
            return;
        }
        String cmd = sc.next();
        if (cmd.equals("add") || cmd.equals("enqueue"))
        {
            String item = sc.next();
            queue.enqueue(item);
            refresh(queue.toString());
            return;
        }
        if (cmd.equals("remove") || cmd.equals("dequeue"))
        {
            queue.dequeue();
            refresh(queue.toString());
            return;
        }
    }

    /**
       The refresh method stores the current queue
       entries into the corresponding text fields of
       the queue view.

     * @param qStr encodes the current contents of the queue.
     */
    private void refresh(String qStr)
    {
        Scanner sc = new Scanner(qStr);
        sc.nextLine();    // Skip first, rear info
        while (sc.hasNext())
        {
            int k = sc.nextInt();
            String qEntry = sc.next();
            qViewTextField[k].setText(qEntry);
        }
    }
}
```

## 20.6 Linked List Implementation of Queues

**CONCEPT: Singly linked lists can be used to implement queues.**

A simple linked list can be used to implement a queue. A reference *front* is used as the head of the linked list, and marks the end of the list from which items will be removed. A second reference, *rear*, is used to mark the end at which additions will take place. Figure 20-7 shows a queue with three elements.

**Figure 20-7** A linked list representing a queue with three elements



Notice that *front* and *rear* will be pointing to the same node whenever there is only one node in the list, and will both be set to `null` whenever the list is empty.

### Queue Initialization and Enqueuing of Items

Using the same `Node` class as in the `LinkedStack` program, we can define a class `LinkedQueue` that has two fields, `front` and `rear`, to keep track of the list of queue items:

```
Node front = null;
Node rear = null;
```

To add an item `s` to an empty queue, we create a node containing `s` and set both `rear` and `front` to point to it as follows:

```
rear = new Node(s, null);
front = rear;
```

To add an item `s` to a nonempty queue, we create a node containing `s`, set `rear.next` to point to that node, and then move `rear` so that it points to the newly added node as follows:

```
rear.next = new Node(s, null);
rear = rear.next;
```

## Dequeuing Items

To remove and return a value from the queue, first we store the value to be returned. Then we move `front` down past the node being removed and return the stored value as follows:

```
String value = front.value;
front = front.next;
return value;
```

This code, however, will leave `rear` pointing to an item that has been removed if the item being removed is the last one in the list. In this case, `rear` must be set to `null`. The correct code for removing is therefore

```
String value = front.value;
front = front.next;
if (front == null)
   rear = null;
return value;
```

The code that implements the `LinkedQueue` class, with appropriate driver code is given in Code Listing 20-10.

**Code Listing 20-10** (`LinkedQueue.java`)

```
/**
   This class implements a queue based
   on linked lists.
*/

public class LinkedQueue
{
   private class Node
   {
      String value;
      Node next;
      Node(String val, Node n)
      {
         value = val;
         next = n;
      }
   }

   private Node front = null;
   private Node rear = null;

   /**
      The method enqueue adds a value
      to the queue.
      @param s The value to be added
      to the queue.
   */

   public void enqueue(String s)
   {
      if (rear != null)
      {
         rear.next = new Node(s, null);
         rear = rear.next;
      }
      else
      {
         rear = new Node(s, null);
         front = rear;
      }
   }

   /**
      The empty method checks to see if
      the queue is empty.
      @return true if and only if queue
      is empty.
   */

   public boolean empty()
   {
      return front == null;
   }

   /**
      The method peek returns value at the
      front of the queue.
      @return item at front of queue.
      @exception EmptyQueueException When the
      queue is empty.
   */

   public String peek()
   {
      if (empty())
         throw new EmptyQueueException();
      else
         return front.value;
   }

   /**
      The dequeue method removes and returns
      the item at the front of the queue.
      @return item at front of queue.
      @exception EmptyQueueException When
      the queue is empty.
   */

   public String dequeue()
   {
      if (empty())
         throw new EmptyQueueException();
      else
      {
         String value = front.value;
         front = front.next;
         if (front == null) rear = null;
         return value;
      }
   }

   /**
      The toString method concatenates all strings
      in the queue to give a string representation
      of the contents of the queue.
      @return string representation of this queue.
   */

   public String toString()
   {
      StringBuilder sBuilder = new StringBuilder();

      // Walk down the list and append all values
      Node p = front;
      while (p != null)
      {
         sBuilder.append(p.value + " ");
         p = p.next;
      }
      return sBuilder.toString();
   }
}
```

Code Listing 20-11 demonstrates the use of the LinkedQueue class.

**Code Listing 20-11** (`LinkedQueueDemo.java`)

```java
/**
  The LinkedQueueDemo class demonstrates
  the use of the LinkedQueue class.
*/

public class LinkedQueueDemo
{
   public static void main(String [] args)
   {
      LinkedQueue queue = new LinkedQueue();

      // Add 4 names
      String [ ] names =
         {"Alfonso", "Bob", "Carol", "Deborah"};
      System.out.println("Adding names: ");
      for (String s : names)
      {
         System.out.print(s + " ");
         queue.enqueue(s);
      }

      System.out.println("\nState of queue is: ");
      System.out.println(queue);

      // Remove 2 names
      System.out.println("Removing 2 names.");
      queue.dequeue(); queue.dequeue();
      System.out.println("State of queue is: ");
      System.out.println(queue);

      // Add another name
      System.out.println("Adding the name Elaine:");
      queue.enqueue("Elaine");
      System.out.println("State of queue is: ");
      System.out.println(queue);
   }
}
```

**Program Output**

```
Adding names:
Alfonso Bob Carol Deborah
State of queue is:
Alfonso Bob Carol Deborah
Removing 2 names.
```

```
State of queue is:
Carol Deborah
Adding the name Elaine:
State of queue is:
Carol Deborah Elaine
```

**Checkpoint**

20.6 What is the common name for an operation that adds an element to a queue?

20.7 In what order are elements added and removed from a queue?

20.8 Cite an example of a queue from everyday life.

20.9 In an array implementation of a queue, why is it necessary to treat the array as a circular buffer?

## 20.7 Generic Implementation of Stacks and Queues

**CONCEPT: Classes that implement collections should be written to use generic types.**

A generic collection class, whether a stack or a queue, has a great advantage: it can be used to hold items of any type. Implementing a generic collection is not much different from implementing a type-specific one: you simply parameterize the name of the class with a type parameter, say `T`, and then use `T` wherever you would need to use the type of the items being stored in the collection. For example, a skeleton for an array implementation of a generic stack class looks like this:

```
class GenStack<T>
{
   private T [] s;      // Holds stack elements
   private int top;     // Stack top pointer
   public GenStack (int capacity)
   {

   }
   public boolean empty() { return top == 0; }
   public T push(T x)
   {

   }
   public T pop()
   {

   }
}
```

There is a complication, however, because Java does not allow arrays of generic type to be instantiated. For example, the following statement to instantiate an array to hold the stack items will not compile:

```
T[ ] s = new T[capacity];
```

There is a workaround: first instantiate an array of `Object`, and then cast it to the desired generic type as follows:

```
T[ ] s = (T[ ]) new Object[capacity];
```

This compiles, albeit with a compiler warning. Code Listing 20-12 shows a complete implementation of an array-based generic stack. The code uses the exception classes declared in the *StackExceptions.java* encountered earlier.

**Code Listing 20-12** (`GenStack.java`)

```
/**
   This class implements a generic array
   based stack.
*/

public class GenStack<T>
{
   private T [] s;   // Body of stack
   private int top;  // Stack top pointer

   /**
      Constructor.
      @param capacity The capacity of the stack.
   */

   public GenStack (int capacity)
   {
      s = (T[ ]) new Object [capacity];
      top = 0;
   }
   /**
      The empty method checks to see if
      the stack is empty.
      @return true if and only if the
      stack is empty.
   */

   public boolean empty() { return top == 0; }

   /**
      The push  method adds x to the stack.
      @param x the value to be pushed onto
      the stack.
      @return the value that was pushed
      onto the stack.
      @exception StackOverFlowException When
      the stack is full.
   */

   public T push(T x)
   {
      if (top == s.length)
         throw new StackOverFlowException();
      else
      {
         s[top] = x;
         top ++;
         return x;
      }
   }

   /**
      The pop method removes and returns the
      item at the top of the stack.
      @return item at the top of the stack.
      @exception EmptyStackException When the
      stack is empty.
   */

   public T pop()
   {
      if (empty())
         throw new EmptyStackException();
      else
      {
         T retVal = s[top-1];
         s[top-1] = null;
         top--;
         return retVal;
      }
   }
}
```

The program in Code Listing 20-13 shows how to use the `GenericStack` class.

**Code Listing 20-13** (GenericStackDemo.java)

```java
/**
   This class demonstrates the use of
   the generic stack class GenStack
*/

public class GenericStackDemo
{
   public static void main(String [] arg)
   {
      GenStack <String> st = new GenStack<String>(5);
      st.push("George");
      st.push("Washington");
      System.out.println(st.pop());
      System.out.println(st.pop());
   }
}
```

**Program Output**

```
Washington
George
```

## 20.8 Queues and Breadth-First Search

You do not need to implement your own queue if you need to use one in your program. The Java Collection Framework provides many different classes that implement the `Queue` interface. One such class is the familiar `LinkedList` class that we encountered in Chapter 19. The `Queue` interface declares the methods shown in Table 20-2.

**Table 20-2** Some methods in the Queue interface

| | |
|---|---|
| `boolean add(E, e)` | Adds an element to the rear of the queue (*enqueue*) |
| `E peek()` | Returns, but does not remove, the element at the front of the queue |
| `E remove()` | Removes and returns the element at the front of the queue (*dequeue*) |
| `boolean isEmpty()` | Checks to see if the queue is empty |

## In the Spotlight:

## Directory Searching Using Breadth-First Search

Queues are very useful in solving problems that involve *breadth-first search*, a type of search through the nodes of a digraph. (Digraphs were introduced in the Spotlight section of Chapter 19.) Breadth-first search starts searching at one node and then continues the search through other nodes in order of increasing distance from the start node. Here we show how breadth-first search can be used to search a directory and all contained subdirectories for a specified file or subdirectory. It is not necessary to understand what a digraph is to follow this presentation.

Files and directories in a computer system are stored in a hierarchical fashion, in which a directory may contain files and other directories. When a directory *D* is contained in another directory *E*, we say that *D* is a child directory of *E*. This creates a situation in which directories can have children, children of children, and, in general, *descendants*. The problem here is to search through some initial directory and all its descendants to find a specified file *F* and print the path to *F* if it is found.

Breadth-first search is based on using a queue to store directories that are descendants of the initial directory. A descendant directory is placed on the queue if it has been encountered, but has not yet been searched. Initially, the queue contains only one directory: the initial directory. We proceed in stages. At each stage, we remove the directory *X* at the front of the queue and search it. If we find the desired file, we stop the search. Otherwise, we add all the child directories of *X* to the (rear of the) queue.

Suppose that the initial directory is *X*. In the beginning, the queue contains only *X*, and *X* will be removed from the queue and searched. If the process does not stop there, then all the children of *X* will be added to the queue. The children of *X* are the first-generation descendants. At the second stage, one of the first-generation descendants, say *Y*, will be removed and searched. If the desired file is not found, then all the children of *Y* (which are second-generation descendants of *X*) will be added to the rear of the queue for later processing. Because nodes will be removed from the queue and searched in the order in which they are added, all directories will be searched in increasing order of their generation. You can see from this that breadth-first search searches nodes in order of increasing distance from the start node.

We need several methods from the `java.util.File` class, as shown in Table 20-3.

**Table 20-3** Some methods of the `File` class

| | |
|---|---|
| `String [] list()` | Returns an array of strings representing names of the files or directories stored in this directory. Returns `null` if the `File` object is not a directory or if there is an IO error in trying to determine the directory listing. |
| `File [] listFiles()` | Returns an array of `File` objects representing the files or subdirectories stored in this directory. Returns `null` if the `File` object is not a directory or if there is an IO error in trying to determine the directory listing. |
| `boolean isDirectory()` | Returns `true` if this `File` object is a directory and not a regular file. |
| `String getAbsolutePath()` | Returns the full pathname of the file or directory. |

The program in Code Listing 20-14 follows closely the preceding description of breadth-first search. For brevity, the test data is hard coded into the main method, and the program searches for a file named “`Menu.tex`” starting in the directory “`C:/Users/gcm`.” As shown by the program output, this file is found in a descendant subdirectory named “`Backup\CSC531`.”

**Code Listing 20-14** (`DirSearch.java`)

```
/**
   This program shows how to use breadth-first search
   based on a queue to do a directory search.
*/
import java.io.File;
import java.util.*;

public class DirSearch
{
    public static void main(String[] args)
    {
       File initDir = new File("C:/Users/gcm");
       String filePath = search(initDir, "Menus.tex");
       if (filePath == null)
           System.out.println("Not found");
       else
           System.out.println(filePath);
    }

    /**
       This method searches a given directory and all its
       subdirectories looking for specified file or subdirectory.
       @param initDir : the initial directory to search.
       @param searchFileName : the name of the file or subdirectory
                               to search for.
       @return the full path name of the searched for file or directory.
    */
    static String search(File initDir, String searchFileName)
    {
      Queue<File> directoriesToSearch = new LinkedList<File>();
      directoriesToSearch.add(initDir);

      while (!directoriesToSearch.isEmpty())
      {
          // Get next directory to search
          File currDir = directoriesToSearch.remove();
          // Get contents of current directory
          String[] dirContents = currDir.list();
          // Directory contents will be null if there is
```

```
     // a problem listing the directory contents
     if (dirContents == null) continue;
     // Do the directory contents contain the desired file?
     if (Arrays.asList(dirContents).contains(searchFileName))
     {
       return currDir.getAbsolutePath();
     }
     // Desired file not in current directory
     // Add all the children of this directory to the queue
     // of directories to be searched
     File [] childDirectories = currDir.listFiles();
     for (File f : childDirectories)
     {
       if (f.isDirectory())
         directoriesToSearch.add(f);
     }
   }
   // No more directories left to search, so not found
   return null;
  }
}
```

**Program Output**

```
C:\Users\gcm\Backup\CSC531
```

## 20.9 Common Errors to Avoid

- **Forgetting to check for an empty stack or queue.** You should always check to see if a stack is empty before calling `peek` or `pop`, and you should always check to see if a queue is empty before attempting to access or dequeue an element from the queue. If you forget to do this, the exception thrown will terminate your program.
- **Handling an empty stack or queue by catching the exception thrown.** Exception handling should be reserved for conditions your program cannot anticipate and prevent. Use the methods for checking if a collection is empty instead.
- **Not maintaining the stack or queue invariant.** If you are writing a stack or queue, establish a convention (invariant) for how you use the fields of your stack or queue class, and make sure all methods preserve the invariant.

## Review Questions and Exercises

### Multiple Choice and True/False

1. A collection that is accessed in *first-in-first-out* fashion is called __________.
   a. a stack
   b. a queue
   c. a linked list
   d. an array-based collection
2. A collection that is accessed in *last-in-first-out* fashion is called ___________.
   a. a stack
   b. a queue
   c. a linked list
   d. none of the above
3. The order in which cars go through a toll booth is best described as __________.
   a. a stack
   b. a queue
   c. a linked list
   d. none of the above
4. The concept of seniority, which some employers use to hire and fire workers is __________.
   a. a stack
   b. a queue
   c. a linked list
   d. none of the above
5. The stack method that returns an element from the stack without removing it is __________.
   a. `pop`
   b. `push`
   c. `peek`
   d. `spy`
6. If the stack method `push` is called on an empty stack, __________.
   a. it throws an `EmptyStackException`
   b. it adds its argument to the stack
   c. it calls the stack method `empty`
   d. none of the above
7. **True or False:** You can use the JCF stack to directly create a stack of `int`.
8. **True or False:** If `pop` is called on an empty stack, it will not return until the user puts something on the stack.
9. **True or False:** When using an array to implement a stack, the `push` method will wrap around to the beginning of the stack when it reaches the end.
10. **True or False:** In a linked implementation of a queue, the references `front` and `rear` can only be equal if the queue is empty.

## Find the Error

Find the error in each of the following code segments:

1.
```
// An array implementation of a stack
int pop()
{
   if (top == 0)
      throw new EmptyStackException( );
   else
   {
      return s[top-1];
      top --;
   }
}
```

2.
```
// A linked implementation of a queue
void enqueue(int x)
{
   if (empty())
   {
      rear = new Node(x);
      front = rear;
   }
   else
   {
      rear = new Node(x);
      rear = rear.next;
   }
}
```

3.
```
//A linked implementation of a stack
int pop()
{
   if(empty())
      throw new EmptyStackException();
   return top.value;
}
```

4.
```
// A linked implementation of a queue
int dequeue()
{
   if(empty())
      throw new EmptyQueueException();
   int value = front.value;
   front ++;
   return value;
}
```

5.
```
// An array implementation of a queue
int dequeue()
{
   if (empty())
      throw new EmptyQueueException();
   int val = q[rear];
   rear ++;
   return val;
}
```

### Algorithm Workbench

1. A palindrome is a word that reads the same backward as forward. For example, the words *madam*, *radar*, *dad*, and *kayak* are all palindromes. Write a method that takes a parameter `s` of type `String` and uses a stack to see if `s` is a palindrome.
2. Write a class that uses an array to implement a stack of integers. The stack should have the following fields:

```
int [ ]s;          // Holds stack elements
int top = -1;      // Points to last item pushed
```

   The field `top` will point to an item that is actually stored on the stack, instead of pointing to the next available slot. The stack will set `top` to −1 when it is empty. Write the constructor for this class, and all the usual stack methods.
3. Suppose that you have two stacks but no queues. You have an application that needs to use a queue. Explain how to use the two stacks to simulate a single queue.

### Short Answer

1. What is the name for a last-in-first-out collection?
2. What is the name for a first-in-first-out collection?
3. What is the name for a first-in-last-out collection?
4. What does the `pop` method do when it is called on an empty stack?
5. Why does the array entry containing an item that is being popped have to be set to `null`?
6. What problem would you encounter if you tried to use the head of a linked list for the rear of a queue, and use the other end of the list for the front?

## Programming Challenges

### 1. Double-Ended Queue

A *deque* (pronounced "deck") is a list-based collection that allows additions and removals to take place at both ends. A deque supports the operations *addFront*(*x*), *removeFront*( ), *addRear*(*x*), *removeRear*( ), *size*( ), and *empty*( ). Write a class that implements a deque that stores strings using a doubly linked list that you code yourself. Demonstrate your class with a graphical user interface that allows users to manipulate the deque by typing appropriate commands in a `JTextField` component, and see the current state of the deque displayed in a `JTextArea` component. Consult the documentation for the `JTextArea` class for methods you can use to display each item in the deque on its own line.

### 2. Array-Based Deque

VideoNote
Array-Based Deque

Implement a deque as described in Programming Challenge 1, except base your implementation on an array. The constructor for the class should accept an integer parameter for the capacity of the deque and create an array of that size. Use a graphical user interface based on an array of text fields similar to what is shown in Figure 20-6. Test your deque class by constructing a deque with capacity 10.

### 3. Prefix Expressions

An expression is in *prefix form* when operators are written before their operands. Here are some examples of prefix expressions and the values they evaluate to:

| Expression | Value |
|---|---|
| 12 | 12 |
| + 2 51 | 53 |
| * 5 7 | 35 |
| * + 16 4 + 3 1 | 80 |

An expression (such as 12) that begins with an integer is a prefix expression that evaluates to itself. Otherwise, an expression is a prefix expression if it begins with an operator and is followed by two prefix expressions. In this latter case, the value of the expression is recursively computed from the values of its constituent prefix sub-expressions.

Write a program that allows the user to enter prefix expressions in a text field. The program reads the expression, evaluates it, and displays the value in a suitable GUI component. Assume that the user enters expressions that use only positive integers and the two operators + and *. Your program should use a stack to store values of sub-expressions as they are computed, and another stack to store operators that have not yet been applied.

### 4. Properly Nested Delimiters

A Java program can have the following type of delimiters: `{`, `}`, `(`, `)`, `[`, and `]`. In a correct Java program, these delimiters must be properly nested. Think of each left delimiter `{`, `(`, and `[` as opening a scope, and think of each right delimiter `}`, `)`, and `]` as closing a scope opened by a corresponding left delimiter. A string of characters containing these delimiters *has proper nesting of delimiters* if each scope that is opened is eventually closed, and the scopes are opened and closed in a last-opened-first-closed fashion.

Write a program that reads a file containing Java source code and checks it for proper nesting of delimiters. Your program should read the source code from the file and print it to the screen. If the file is properly nested, all of it is printed to the screen and a message is printed that the file is properly nested. If the file is not properly nested, then copying of the file to the screen stops as soon as improper nesting is detected, and your program prints a message that the file has errors. To simplify your task, you may assume these delimiters do not appear inside of comments and string literals, and that they do not appear in the program as character literals.

### 5. Tracing Genealogies

A file has genealogy data for a collection of *N* people. The first line of the file contains the integer *N* followed by *N* additional lines of data. Each of these additional lines specifies a list of children for a single person. The line starts with the name of the person, followed by the number of that person's children, followed by the names of the children. Here is an example of a file specifying genealogy information for 10 people.

```
10
Al     3   Beth Carol Dino
Beth   1   Pablo
Carol  3   Ben Alex Mary
Dino   0
Pablo  2   Angela Miguel
Ben    0
Alex   0
Mary   0
Angela 0
Miguel 0
```

For example, Al has three children named Beth, Carol, and Dino; Beth has one child named Pablo; and Dino has no children. You may assume that all names are unique.

Write a program which reads a file of genealogy information and then allows the user to enter pairs of names *X* and *Y*. The program then determines whether *Y* is a descendant of *X*, and if so, prints a list of names beginning with *X* and ending with *Y*, such that each person in the chain is a child of person preceding them on the list. Otherwise, the program states that *Y* is not a descendant of *X*.

CHAPTER

# 21

# Binary Trees, AVL Trees, and Priority Queues

**TOPICS**

21.1 Binary Trees and Their Applications
21.2 Binary Search Trees
21.3 AVL Trees
21.4 Priority Queues
21.5 Common Errors to Avoid

## 21.1 Binary Trees and Their Applications

**CONCEPT: A binary tree is like a list in which every node can have up to two successors.**

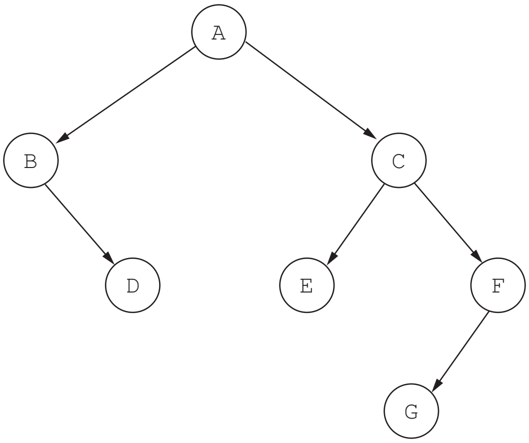
A *binary tree* is a collection of nodes in which each node is associated with up to two successor nodes, respectively called the *left* and *right child*. Not every node in a binary tree will have two children: the left child may be missing, or the right child may be missing, or both nodes may be missing. A node in a binary tree that has no children is called a *leaf*.

A node that has children is said to be the *parent* of its children. For a nonempty collection of nodes to qualify as a binary tree, every node must have at most one parent, and there must be exactly one node with no parent. The one node that has no parent is called the *root* of the binary tree. An empty collection of nodes is regarded as constituting an empty binary tree.

A binary tree is similar in some ways to a linked list: the root of the binary tree corresponds to the head of the list, a child of a node in a binary tree corresponds to the successor of a node in a list, and the parent of a node in a binary tree corresponds to the *predecessor* of a node in a list. And of course, the analogue of the empty list is the empty binary tree.

Figure 21-1 shows an example of a binary tree. The node labeled *A* is the root, and each arrow points from a parent to a child.

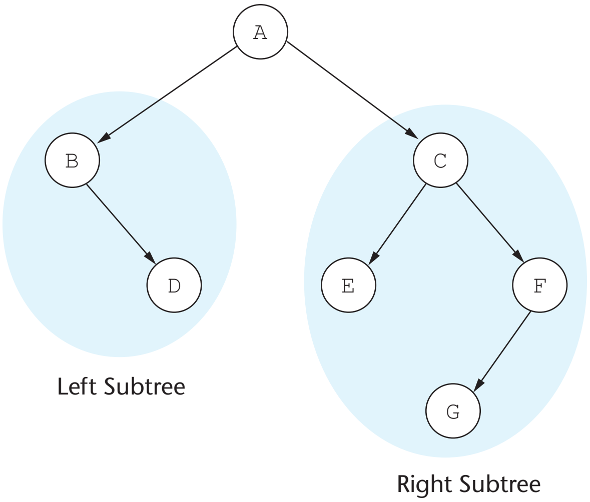
**Figure 21-1** A binary tree



## Binary Tree Concepts

Let *X* be a node in a binary tree *T*. A node *Y* is a *descendant* of *X* if *Y* is on a path from *X* to a leaf of *T*. Thus, the descendants of *X* are *X* itself, all the children of *X*, all the children of the children of *X*, and so on. In Figure 21-1, the descendants of *C* are *C*, *E*, *F*, and *G*. Any node *X* together with all its descendants forms a binary tree with root *X*: such a binary tree is called a *subtree* of *T*. In particular, the set of all descendants of the left child of the root of *T* forms a binary tree called the *left subtree* of *T*. Likewise, the set of all descendants of the right child of the root of *T* comprises the *right subtree* of *T*. Figure 21-2 illustrates the left and right subtrees of the binary tree of Figure 21-1.

**Figure 21-2** Left and right subtrees



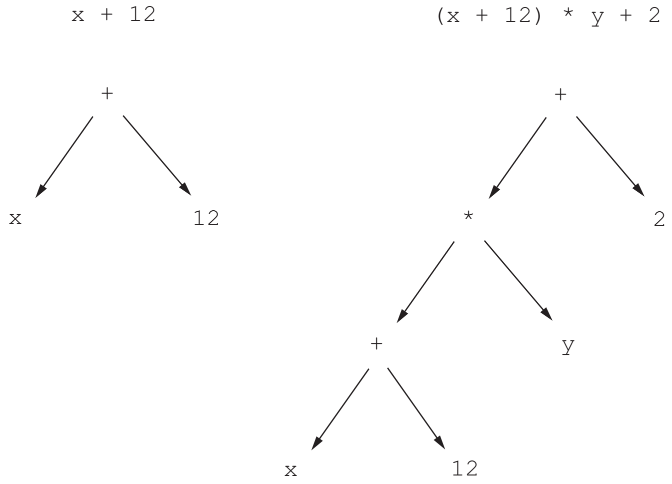
Although it is an abuse of language, we will sometimes speak of a subtree rooted at a child of a node *X* as being a subtree of *X*. According to this, {*B*, *D*} is the left subtree of *A*, and {*F*, *G*} is the right subtree of *C*.

An *ancestor* of *X* is any node on the path from *X* to the root of *T*. The ancestors of *F* in the binary tree of Figure 21-1 are *F*, *C*, and *A*.

## Applications of Binary Trees

Binary trees and their generalizations have many applications. They can be used to organize information in ways that allow efficient search and retrieval. Other forms of trees closely related to binary trees are used in database management systems to represent data, and in programming language compilers and interpreters to represent expressions. Figure 21-3 shows the use of binary trees to represent arithmetic expressions.

**Figure 21-3** Binary tree representation of expressions



## Representing Nodes of Binary Trees

A node in a binary tree is an object that contains a field for storing a value and two other fields for storing references to the left and right children. Here is a class that can be used to create nodes for a binary tree that stores integer data:

```
class Node
{
   int value;          // Value to be stored
   Node left, right;   // Left and right children
   Node(int val)
   {
      value = val;
      left = null;
      right = null;
   }
```

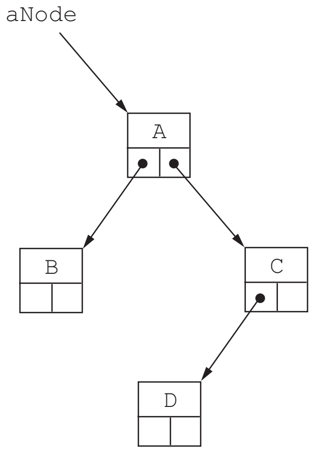
```
      Node(int val, Node left1, Node right1)
      {
         value = val;
         left = left1;
         right = right1;
      }
   }
```

Notice the two constructors in the `Node` class. The constructor with one parameter is convenient when we need to create leaf nodes.

A binary tree is represented by a reference to its root node. Assuming we have integer values A, B, C, and D, we can build the binary tree shown in Figure 21-4 by executing the following sequence of statements:

```
Node dNode = new Node(D);
Node cNode = new Node(C, dNode, null);
Node bNode = new Node(B);
Node aNode = new Node(A, bNode, cNode);
```

**Figure 21-4** A binary tree constructed with the Node class



## Traversing a Binary Tree

A *binary tree traversal* is a systematic method of doing some processing at every node of a binary tree. A common reason for traversing a binary tree is to print the values stored in it. There are three standard techniques for traversing binary trees. Each of these is most easily implemented as a recursive method called on the root node of the binary tree. They are as follows:

- ***Preorder traversal***: process the data at the root node, traverse the left subtree, and then traverse the right subtree
- ***Inorder traversal***: traverse the left subtree, process the data at the root node, and then traverse the right subtree
- ***Postorder traversal***: traverse the left subtree, traverse the right subtree, and then process the data at the root node

Processing the data at a node is often called *visiting* the node.

A traversal method takes as parameter a reference `btree` to the root of a subtree. The method returns immediately if `btree` is `null`, otherwise, it visits the root and recursively traverses the two subtrees in the order prescribed by the traversal technique being used. Here is an implementation of inorder traversal that prints out all values stored in binary tree:

```
void inorder(Node btree)
{
   // If the tree is empty do nothing
   if (btree != null)
   {
      // Traverse the left subtree
      inorder(btree.left);
      // Visit the node
      System.out.print(btree.value + " ");
      // Traverse the right subtree
      inorder(btree.right);
   }
}
```

Code Listing 21-1 shows a `Node` class that can be used as a basis for building a binary tree that stores integer values.

**Code Listing 21-1** (`Node.java`)

```
/**
   Node class.
*/
public class Node
{
    int value;
    Node left, right;

    // Constructor for leaf nodes.
    Node(int val)
    {
        value = val;
        left = null;
        right = null;
    }

    // Constructor for non-leaf nodes.
    Node(int val, Node leftChild, Node rightChild)
    {
        value = val;
        left = leftChild;
        right = rightChild;
    }
}
```

Code Listing 21-2 shows a class called `NodeUtilities`. This class can be used as a repository for various methods for working with node-based structures.

**Code Listing 21-2** (`NodeUtilities.java`)

```
/**
  This class has various utility methods for working with nodes.
*/
public class NodeUtilities
{
    /**
      Inorder traversal of a binary tree rooted at a node.
      @param btree : The root of the binary tree.
    */
    static public void inorder(Node btree)
    {
        if (btree != null)
        {
            inorder(btree.left);
            System.out.print(btree.value + " ");
            inorder(btree.right);
        }
    }
}
```

Code Listing 21-3 uses these two classes. The program creates a collection of nodes, links them together to form a binary tree, and then traverses the binary tree in inorder.

**Code Listing 21-3** (`BinaryTreeNodes.java`)

```
/**
  This program demonstrates construction and traversal of binary trees.
*/

public class BinaryTreeNodes
{
    // Creates a binary tree from nodes and traverses it in
    // in inorder.
    public static void main(String[] args)
    {
        Node root = null;  // Will be root of the binary tree.

        Node aNode = new Node(10);
        aNode.left = new Node(20);
        Node dNode = new Node(40);
        Node cNode = new Node(30, dNode, new Node(50));
        aNode.right = cNode;
        root = aNode;

        System.out.print("Inorder traversal is: ");
        NodeUtilities.inorder(root);
        System.out.println();
    }
}
```

**Program Output**

```
Inorder traversal is: 20 10 40 30 50
```

**Checkpoint**

21.1 Consider the binary tree *T* shown in Figure 21-1.

- a) List the ancestors of each node of *T*.
- b) List the descendants of each node of *T*.
- c) List all the leaves of *T*.
- d) List all the non-leaf nodes of *T*.
- e) Show the nodes of *T* as output by a preorder traversal.
- f) Show the nodes of *T* as output by a postorder traversal.
- g) Show the nodes of *T* as output by an inorder traversal.

21.2 Show the output from a postorder traversal of the two trees in Figure 21-3.

21.3 Show the output from a preorder traversal of the two trees in Figure 21-3.

21.4 Using the `Node` class defined in this section, draw the binary tree constructed by the following statements:

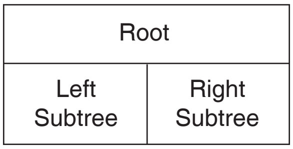
```
Node left = new Node(10, new Node(20), new Node(50));
Node right = new Node(60, new Node(180), null);
Node tree = new Node(160, left, right);
```

and list the nodes of the resulting binary tree in preorder.

## Graphical Display of Binary Trees

You can graphically display a binary tree with the same recursive logic that is used to traverse the tree. The main idea is to recursively partition a view into three sections as shown in Figure 21-5.

**Figure 21-5** Partitioning a view to display a binary tree



The partitioning is done by nesting views within views. An empty binary tree is displayed using a blank view. If the tree to be displayed is not empty, the display view is partitioned as in Figure 21-5, the root is displayed at the top, and the left and right subtrees are recursively displayed in subviews nested side by side in the lower part of the containing view.

Code Listing 21-4 shows a `GUINodeUtilities` class that has a recursive method for displaying a graphical view of a binary tree rooted at a node. If the tree is empty, its graphical representation is a blank `HBox` pane. A leaf node is displayed as a `HBox` containing a single un-editable text field that displays the value in the leaf node. A nonleaf node with a value *v* is displayed using a `BorderPane`: the value *v* is displayed in the TOP region, while the LEFT and RIGHT regions recursively display the left and right subtrees.

**Code Listing 21-4** (`GUINodeUtilities.java`)

```java
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;

/**
   This class has a utility method for the
   graphical display of a binary tree.
*/

class GUINodeUtilities
{
    public static Pane getView(Node root)
    {
        // Empty tree is displayed as an empty  HBox
        if (root == null) { return new HBox(); }

        // Leaf node displayed as a single un-editable text field
        // in a HBox.
        if (root.left == null && root.right == null)
        {
            TextField tF = new TextField(String.valueOf(root.value));
            tF.setPrefColumnCount(2);
            tF.setEditable(false);
            HBox hBox = new HBox();
            hBox.setAlignment(Pos.CENTER);
            hBox.getChildren().add(tF);
            hBox.setPadding(new Insets(5));
            return hBox;
        }
        // Non-leaf node uses a BorderPane
        // that puts the value in the TOP region, and left and right
        // subtrees in the LEFT and RIGHT regions.
        BorderPane pane = new BorderPane();
        pane.setStyle("-fx-border-color: gray");
        pane.setPadding(new Insets(10));

        // Value of a non-leaf node is displayed in a text field
        // in a HBox in the TOP region of the BorderPane.
        String value = String.valueOf(root.value);
        TextField tF2 = new TextField(value);
        tF2.setPrefColumnCount(2);
        tF2.setEditable(false);
        HBox valueHBox = new HBox();
        valueHBox.setAlignment(Pos.CENTER);
        valueHBox.getChildren().add(tF2);
        valueHBox.setPadding(new Insets(5));
        pane.setTop(valueHBox);

        // Left and right subtrees are recursively displayed
        // in the LEFT and RIGHT region of the BorderPane.
        pane.setLeft(getView(root.left));
        pane.setCenter(new Label("         ")); // For spacing.
        pane.setRight(getView(root.right));

        return pane;
    }
}
```

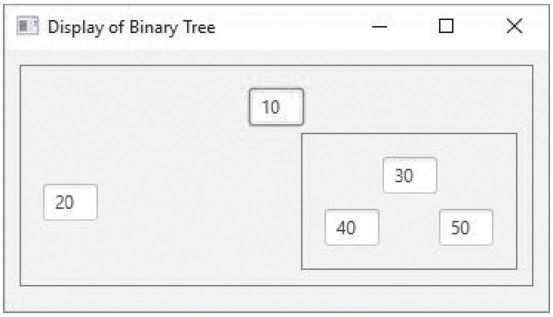
Code Listing 21-5 shows a program that uses this utility method to display a binary tree in a scene on a JavaFX stage. The result of running the program can be seen in Figure 21-6.

**Code Listing 21-5** (`GUIBtreeDisplayDemo.java`)

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

/**
```

```
   This program demonstrates the GUI display of a
   binary tree.
*/

public class GUIBtreeDisplayDemo extends Application
{
    public static void main(String [] args)
    {
       launch();
    }

    @Override
    public void start(Stage stage) throws Exception
    {
        // Create the tree.
        Node root;
        Node aNode = new Node(10);
        aNode.left = new Node(20);
        Node dNode = new Node(40);
        Node cNode = new Node(30, dNode, new Node(50));
        aNode.right = cNode;
        root = aNode;

        // Display the binary tree in a scene on the stage.
        VBox vBox = new VBox(GUINodeUtilities.getView(root));
        vBox.setPadding(new Insets(10));
        Scene scene = new Scene(vBox);
        stage.setScene(scene);
        stage.setTitle("Display of Binary Tree");
        stage.show();
    }
}
```

**Figure 21-6** GUI display of a binary tree (Mark Schrier. Copyright Oracle, Inc.)



## Class Implementation of Binary Trees

As we have seen, a binary tree is implemented as a reference to a root node that is an object of a suitable `Node` class such as shown in Code Listing 21-1. A simple class implementation might use a private nested `Node` class like this:

```
class BinaryTree
{
    private Node {  /* Node fields and methods */ }
    private Node root;
}
```

The class used to represent nodes is usually an implementation detail that should not concern users of the class, so the above approach makes sense. The class could then proceed to add public methods to work with binary tree objects.

Even so, it will be convenient for us to take a different approach. We will not nest the `Node` class, and we will declare the `root` field as *protected* instead of *private*:

```
class Node
{
    /* Node fields and methods */
}
class BinaryTree
{
     protected Node root;
}
```

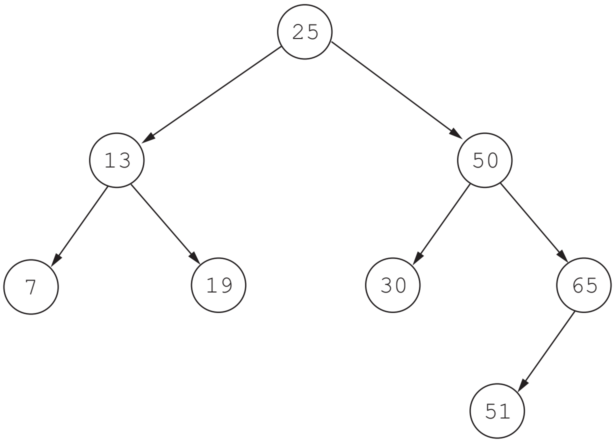
Making `root` protected will make it accessible to subclasses, allowing us to use inheritance to write different types of binary trees. Not nesting Node will allow us to have different binary tree classes that refer to the same `Node` class, thus avoiding unnecessary repetition of code.

## 21.2 Binary Search Trees

**CONCEPT: Binary search trees organize information to facilitate efficient search.**

In Chapter 16, you saw how binary search can be used to search quickly for a given item in a sorted array. Binary search trees are special types of binary trees that can be used to implement a form of binary search that does not require arrays. Look at the binary tree in Figure 21-7.

**Figure 21-7** A binary search tree



Notice that the values stored in the left subtree (7, 13, and 19) are all less than the value stored at the root (25), and that the values stored in the right subtree (30, 50, 51, and 65) are all greater than the value stored in the root. In fact, the tree has the following property: at every node *N*, all values stored in the left subtree of *N* are less than the value stored at *N*, and all values stored in the right subtree of *N* are greater than the value stored at *N*. A binary tree with this property is called a *binary search tree*.

Binary search trees have the potential to facilitate rapid searches. Suppose we want to determine if *X* is in the tree. Starting at the root, we make our way down some path of the binary tree, comparing *X* to each node that we come to. At each node *N*, we continue the search in the left subtree if *X* is less than *N*, or in the right subtree if *X* is greater. We stop when we find *X* at some node, or when we come to a leaf that is not *X*. Assuming that the tree is reasonably balanced, we see that we will discard approximately half of the remaining tree every time we make a comparison. Thus, searching a binary tree with $n$ nodes will require approximately log $n$ comparisons. Naturally, binary search can only be used with items that support a comparison operation with the property that for any two distinct elements $x$ and $y$, either $x$ "is less than" $y$ or $x$ "is greater than" $y$. Sets that support such a comparison operation are said to be *totally* or *linearly* ordered.

The following method searches a binary search tree for an element $x$. It returns `true` if the tree contains $x$ and `false` otherwise.

```
private boolean contains(int x, Node bstree)
{
   if (bstree == null)
      return false;
   if (x == bstree.value)
      return true;
   if (x < bstree.value)
   {
      // Recursively look in left subtree
      return contains(x, bstree.left);
   }
   else
   {
      // Recursively look in right subtree
      return contains(x, bstree.right);
   }
}
```

## Adding a Value to a Binary Search Tree

Like most operations on binary trees, adding a value to a binary search tree has a natural recursive implementation. The method

```
Node add(int x, Node bstree)
```

adds `x` to the binary search tree rooted at `bstree` and returns a reference to the root of the resulting binary search tree. First, the method checks to see if `bstree` is empty: if so, it creates and returns a new node containing `x`. If `bstree` is not empty, the method compares `x` to the element stored in the root: if `x` is less, it adds `x` to the left subtree and makes the resulting tree the new left subtree. If `x` is greater, it adds `x` to the right subtree and makes the resulting tree the new right subtree. Then it returns a reference to the root:

```
private Node add(int x, Node bstree)
{
   if (bstree == null)
      return new Node(x);
   // bstree is not null.
   if (x < bstree.value)
   {
      // Add x to the left subtree and replace the
      // current left subtree with the result
      bstree.left = add(x, bstree.left);
   }
   else
   {
      // Add x to the right subtree
      bstree.right = add(x, bstree.right);
   }
   return bstree;
}
```

## Removing a Value from a Binary Search Tree

**VideoNote**
Removing a Value from a Binary Search Tree

To remove a value from a binary search tree, you must locate the node *N* that contains the value and remove it from the tree. There are three cases to consider:

- The node *N* has no children. Remove *N* by setting the reference to it in its parent node to `null`.
- The node *N* has one child. Remove *N* and promote the child of *N* to take the place of its parent in the tree, as shown in Figure 21-8.

- The node *N* has two children. Remove *N* and replace it with the largest node in its left subtree, as shown in Figure 21-9. Notice that the largest node in the left subtree of *N* cannot have a right child, so removing it reduces it to the simpler case of removing a node with at most one child.

**Figure 21-8** Before and after removal of a node with one child

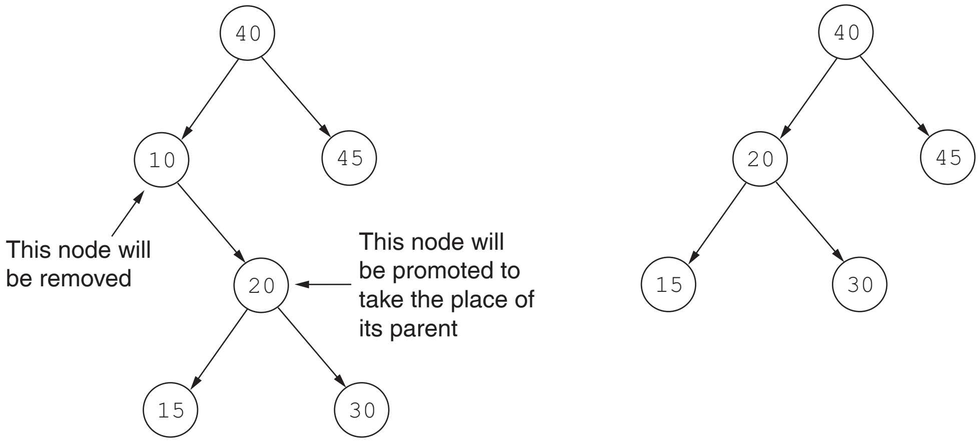
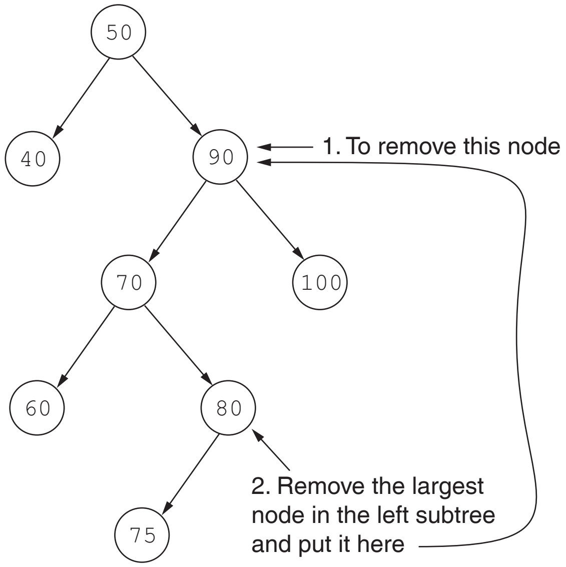
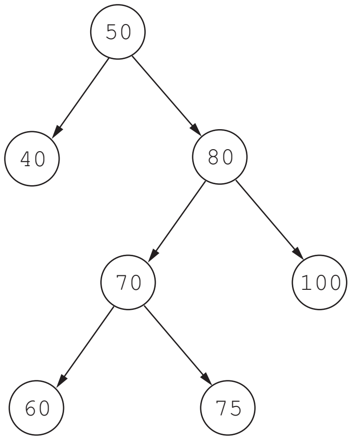

40
10
45
This node will
be removed
20
This node will
be promoted to
take the place of
its parent
15
30
40
20
45
15
30

**Figure 21-9** Removing a node with two children


50
40
90
1. To remove this node
70
100
60
80
75
2. Remove the largest
node in the left subtree
and put it here

The tree that results from removing the node with 90 in the tree of Figure 21-9 is shown in Figure 21-10.

**Figure 21-10** The tree of Figure 21-9 after removing 90



Let us now consider the task of coding the removal of a value *x* from a binary search tree *T*. This operation yields two objects: a node *N* that contains *x*, and the tree that remains after *N* has been removed from *T*. The following class will be used to represent the result of such an operation:

```
private class RemovalResult
{
   Node node;     // Removed node
   Node tree;     // Remaining tree
   RemovalResult(Node node, Node tree)
   {
      this.node = node;
      this.tree = tree;
   }
}
```

This class will be private to the class that represents the binary search tree. The method that performs the actual removal,

```
RemovalResult remove(Node bTree, int x)
```

will take a reference `bTree` to the root of a binary search tree and an integer value `x`. The `remove` method returns `null` if `x` is not found in `bTree`.

The algorithm for removing `x` considers three cases:

1. The relation `x < bTree.value` holds. In this case `x` is recursively removed from the left subtree:

```
RemovalResult result = remove(bTree.left, x);
if (result == null)
   return null;
bTree.left = result.tree;
result.tree = bTree;
return result;
```

If the recursive call returns `null`, then `x` is not in the left subtree and therefore cannot be in the tree, so the original call also returns null. Otherwise, the remaining tree from the recursive call becomes the left subtree of the remaining tree from the original call, and the `result` object is accordingly modified and returned.

2. The relation `x > bTree.value` holds. In this case `x` is recursively removed from the right subtree. This case is a mirror image of the preceding.
3. The relation `x == bTree.value` holds, so that the root node `bTree` contains the value to be removed. In this case, `bTree` becomes the removed node and the remaining tree is assembled from the left and right subtrees of `bTree`. The assembly of the remaining tree differs according to whether `bTree` has zero, one, or two children, and proceeds as previously explained and shown in Figures 21-7, 21-8, and 21-9. The programming details are as follows:

Consider first the case where `bTree` has zero children. The removed node is just `bTree`, and the remaining tree is empty:

```
if (bTree.right == null && bTree.left == null)
   return new RemovalResult(bTree, null);
```

Next, consider the case of `bTree` having one child. In this case `bTree` is the removed node, and the nonempty subtree of `bTree` becomes the remaining tree:

```
Node node = bTree;      // Removed node
Node tree;              // Remaining tree

// Remaining tree is the one nonempty subtree
if (bTree.left != null)
   tree = bTree.left;
else
   tree = bTree.right;
node.left = null;
node.right = null;
return new RemovalResult(node, tree);
```

Finally, consider the case of `bTree` having two children. Again, `bTree` becomes the removed node. The removal of `bTree` leaves two subtrees, and we have to find a way to combine these two subtrees into one binary search tree. This can be done by removing the largest node from the left subtree and using it to replace the removed root. The removal of the largest node from a binary search tree is accomplished by the following recursive method:

```
RemovalResult removeLargest(Node bTree)
```

To see the logic behind the algorithm used by this method, observe that the largest node in a nonempty binary search tree is the root if the root has no right subtree; otherwise, it is the largest node in the right subtree.

```
RemovalResult removeLargest(Node bTree)
{
   if (bTree == null)
      return null;
   if (bTree.right == null)
   {
```

```
            // Root is the largest node
            Node tree = bTree.left;       // Remaining tree
            bTree.left = null;
            return new RemovalResult(bTree, tree);
         }
         else
         {
            // Remove the largest node from the right subtree
            RemovalResult remResult = removeLargest(bTree.right);
            bTree.right = remResult.tree;
            remResult.tree = bTree;
            return remResult;
         }
      }
```

## Implementation of Binary Search Trees

We are now ready to implement a binary tree search class. We begin with a base binary tree class, shown in Code Listing 21-6, that contains a single method for checking whether the tree is empty.

**Code Listing 21-6** (`BinaryTree.java`)

```
/**
   Base binary tree class
*/
public class BinaryTree
{
    protected Node root;

    // Check if the binary tree is empty.
    public boolean isEmpty()
    {
        return root == null;
    }
}
```

A binary search tree is a special type of binary tree, so it makes sense to make the `BinarySearchTree` class a subclass of `BinaryTree`. Code Listing 21-7 shows the implementation of that class, together with all the methods that we have already discussed.

**Code Listing 21-7** (`BinarySearchTree.java`)

```
/**
  This class implements a binary search tree.
*/
```

```

public class BinarySearchTree extends BinaryTree
{
    /**
      This class represents the result of removing
      a node from a binary tree.
    */
    private class RemovalResult
    {
        Node node;    // Removed node
        Node tree;    // Remaining tree

        RemovalResult(Node node, Node tree)
        {
            this.node = node;
            this.tree = tree;
        }
    }

    /**
      The public add method adds a value to the tree by
      calling a private add method and passing it the
      root of the tree.
      @param x The value to add to the tree.
      @return true.
    */
    public boolean add(int x)
    {
        root = add(x, root);
        return true;
    }

    /**
      The contains method checks to see if a value is
      in the binary tree.
      @param x The value to check for.
      @return true if x is in the tree, false otherwise.
    */
    public boolean contains(int x)
    {
        // Call the private recursive method.
        return contains(x, root);
    }

    /**
      The add method adds a value to the search tree.
      @x the value to add.
      @param bstree The root of the binary search tree.
      @return The root of the resulting binary search tree.
    */
    private Node add(int x, Node bstree)
    {
        if (bstree == null)
        {
            return new Node(x);
        }
        // bstree is not null.
        if (x < bstree.value)
        {
            // Add x to the left subtree and replace
            // the current left subtree with the result.
            bstree.left = add(x, bstree.left);
        } else
        {
            // Add x to the right subtree.
            bstree.right = add(x, bstree.right);
        }
        return bstree;
    }

    /**
      The method contains checks whether an item is in
      a binary search tree.
      @param x The item to check for.
      @param bstree The binary tree to look in.
      @return true if found, false otherwise.
    */
    private boolean contains(int x, Node bstree)
    {
        if (bstree == null)
        {
            return false;
        }
        if (x == bstree.value)
        {
            return true;
        }
        if (x < bstree.value)
        {
            // Recursively look in left subtree.
            return contains(x, bstree.left);
        } else
        {
            // Recursively look in right subtree.
            return contains(x, bstree.right);
        }
    }

    /**
      The remove method removes a value from the
      binary search tree.
      @param x The value to remove.
      @returns true if x was removed, false if x not found.
    */
    public boolean remove(int x)
    {
        RemovalResult result = remove(root, x);
        if (result == null)
        {
            return false;
        } else
        {
            root = result.tree;
            return true;
        }
    }

    /**
      This remove method removes a value a from a
      binary search tree and returns the removed node
      and the remaining tree wrapped in a RemovalResult object.

      @param bTree The binary search tree.
      @param x The value to be removed.
      @return null if x is not found in bTree.
    */
    private RemovalResult remove(Node bTree, int x)
    {
        if (bTree == null)
        {
            return null;
        }
        if (x < bTree.value)
        {
            // Remove x from the left subtree.
            RemovalResult result = remove(bTree.left, x);
            if (result == null)
            {
                return null;
            }
            bTree.left = result.tree;
            result.tree = bTree;
            return result;
        }
        if (x > bTree.value)
        {
            // Remove x from the right subtree.
            RemovalResult result = remove(bTree.right, x);
            if (result == null)
            {
                return null;
            }
            bTree.right = result.tree;
            result.tree = bTree;
            return result;
        }
        // x is in this root node.
        // Is it a leaf?
        if (bTree.right == null && bTree.left == null)
        {
            return new RemovalResult(bTree, null);
        }

        // Does the node have two children?
        if (bTree.right != null && bTree.left != null)
        {
            // Remove largest node in left subtree and
            // make it the root of the remaining tree.
            RemovalResult remResult = removeLargest(bTree.left);
            Node newRoot = remResult.node;
            newRoot.left = remResult.tree;
            newRoot.right = bTree.right;

            // Prepare the result to be returned.
            bTree.left = null;
            bTree.right = null;
            return new RemovalResult(bTree, newRoot);
        }
        // The node has one child
        Node node = bTree;
        Node tree;
        if (bTree.left != null)
        {
            tree = bTree.left;
        } else
        {
            tree = bTree.right;
        }
        node.left = null;
        node.right = null;
        return new RemovalResult(node, tree);
    }

    /**
      The removeLargest method removes the largest node
      from a binary search tree.
      @param bTree: The binary search tree.
      @return The result of removing the largest node.
    */
    private RemovalResult removeLargest(Node bTree)
    {
        if (bTree == null)
        {
            return null;
        }
        if (bTree.right == null)
        {
            // Root is the largest node
            Node tree = bTree.left;
            bTree.left = null;
            return new RemovalResult(bTree, tree);
        } else
        {
            // Remove the largest node from the right subtree.
            RemovalResult remResult
                     = removeLargest(bTree.right);
            bTree.right = remResult.tree;
            remResult.tree = bTree;
            return remResult;
        }
    }
}
```

## Graphical Display of Binary Search Trees

The graphical display of binary trees can be achieved by using the `getView()` method of the `NodeUtilities` classes of Code Listing 21-4. The `getView()` method requires access to the root node of the binary tree to be displayed. Because the `root` field of `BinaryTree` is protected, the `getView()` method must be called from a subclass of `BinaryTree`.

What we need is a `DisplayableBtree` class, a subclass of `BinaryTree` that is able to wrap a pre-existing binary tree and produce a graphical view of the wrapped binary tree. The definition of this class starts out like this:

```
public class DisplayableBtree extends BinaryTree
{
    private BinaryTree tree;
    public DisplayableBtree(BinaryTree t)
    {
        tree = t;
    }
}
```

Notice that this class has only one constructor, so you cannot create an instance of `DisplayableBtree` unless you already have an existing binary tree that you want to display:

```
BinarySearchTree  tree1 = new BinarySearchTree();
tree1.add(50);
DisplayableBtree dTree = new DisplayableBTree(tree1);
```

Notice also that `DisplayableBtree` has two fields: a protected field root that it inherits from `Binarytree`, and a private field tree that references another binary tree. So, `DisplayableBtree` is a binary tree that contains, or wraps, another binary tree. Moreover, the inherited `root` field will always be `null`, because unlike `BinarySearchTree`, our BinaryTree class has no add method that can be used to add elements. To make the inherited `isEmpty()` method useful, we override it so that it checks the emptiness of the wrapped binary tree:

```
public class DisplayableBtree extends BinaryTree
{
    private BinaryTree tree;
    public DisplayableBtree(BinaryTree t)
    {
        tree = t;
    }

    @Override
    public boolean isEmpty()
    {
       return tree.isEmpty();
    }
}
```

Finally, we complete the class by adding a `getView()` method to return a graphical view of the binary tree. The completed class is shown in Code Listing 21-8.

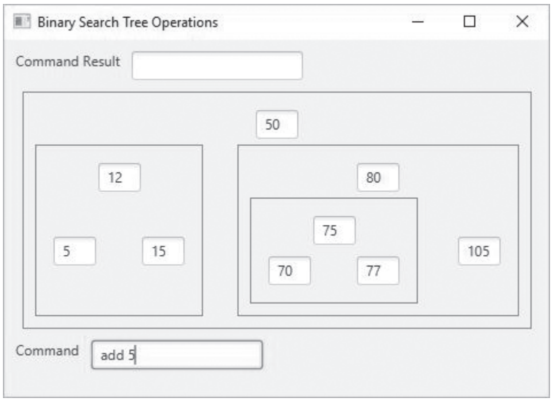
**Code Listing 21-8** **(`DisplayableBtree.java`)**

```
import javafx.scene.layout.Pane;

public class DisplayableBtree extends BinaryTree
{
    private final BinaryTree tree;
    public DisplayableBtree(BinaryTree t)
    {
        tree = t;
    }

    @Override
    public boolean isEmpty()
    {
       return tree.isEmpty();
    }

    // Return a view of the binary tree suitable for
    // display
    public Pane getView()
    {
       return  GUINodeUtilities.getView(tree.root);
    }
}
```

## A Program for Testing the Binary Search Tree Class

The program for testing the class of Code Listing 21-8 will have a graphical user interface that allows users to type commands to add, remove, and search for items in the binary search tree. The user interface consists of a main window with a command entry text field at the bottom and an un-editable text field at the top used to display the results from executing user commands. The center of the window is used to display in a hierarchical tree form, the elements that are currently in the search tree. Figure 21-11 shows the interface after the user has executed commands to add 50, 100, 75, 80, 12, 105, and 77 in succession.

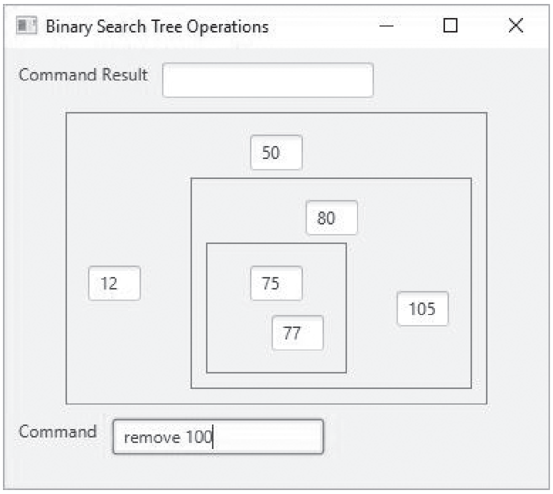
**Figure 21-11** A binary tree after adding some values (Mark Schrier. Copyright Oracle, Inc.)



Command: add 5

Removing the node with 100 causes the program to replace it with 80, resulting in the display shown in Figure 21-12.

**Figure 21-12** The binary tree of Figure 21-11 after removing 100 (Mark Schrier. Copyright Oracle, Inc.)



Command: remove 100

The program allows the following commands:

- `add x`: Adds the number `x` to the binary search tree, for example: `add 12`.
- `remove x`: Removes `x` from the tree, for example: `remove 12`.
- `contains x`: Checks if `x` is in the tree, for example: `contains 12`. The program will display `true` in the command result label at the top of the frame if `x` is found in the tree, and `false` otherwise.
- `isempty`: Checks if there are any values in the tree, for example: `isempty`. The program will display `true` in the command result text field if the tree is empty, and `false` otherwise.

The program for testing the `BinarySearchTree` class follows. It creates a binary search tree on which the `add`, `remove`, and other operations will be performed, and then wraps the search tree with a `DisplayableBtree`:

```
BinarySearchTree binTree = new BinarySearchTree();
DisplayableBtree displayBinTree = new DisplayableBtree(binTree);
```

The program then builds the user interface and attaches an event handler on the command entry text field at the bottom of the window. Every time the user enters a command, the handler method reads the command from the text field and executes the command on the binary search tree `binTree`. The old view of the tree is then removed from the user interface and replaced with the new view. The details are in Code Listing 21-9.

**Code Listing 21-9** (`GUIBSearchTreeDemo.java`)

```
import java.util.Scanner;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

/**
   This class is a GUI interface for testing the
   the binary search tree class.
*/

public class GUIBSearchTreeDemo extends Application
{
    @Override
    public void start(Stage stage) throws Exception
    {
       // Set up the binary tree to manipulate
       BinarySearchTree binTree = new BinarySearchTree();

       // This binary tree wraps the real binary tree
       // so it can be displayed.
       DisplayableBtree
           displayBinTree = new DisplayableBtree(binTree);

       // Displays the result of the user command.
       TextField cmdResultTextField = new TextField();
       cmdResultTextField.setEditable(false);

       // Used by the user to enter a command.
       TextField cmdTextField = new TextField();

       // HBox1 holds cmdResultTextField and its label.
       HBox hBox1 = new HBox(10);
       Label cmdResultLabel = new Label("Command Result");
       hBox1.getChildren().addAll(cmdResultLabel, cmdResultTextField);

       // Place holder for the binary tree view.
       HBox bTreeView = new HBox();
       bTreeView.setAlignment(Pos.CENTER);

       // HBox2 holds the cmdTextField and its label
       HBox hBox2 = new HBox(10);
       Label cmdLabel = new Label("Command");
       hBox2.getChildren().addAll(cmdLabel, cmdTextField);

```

```
        // Put the three HBox objects in a VBox
        VBox outerPane = new VBox(10);
        outerPane.setPadding(new Insets(10));
        outerPane.getChildren().addAll(hBox1, bTreeView, hBox2);

        // Command handler needs to access to the bTreeView
        // so it can set the binary tree view, and to the
        // cmdResultTextField so it can set the command result.
        // Also needs access to the binary tree and its displayable
        // twin.
        EventHandler<ActionEvent>
            handler = new CmdHandler(bTreeView, cmdResultTextField,
                                     binTree, displayBinTree);
        cmdTextField.setOnAction(handler);

        // Set up the stage
        stage.setTitle("Binary Search Tree Operations");
        stage.setScene(new Scene(outerPane));
        stage.show();
    }

    public static void main(String [] args)
    {
         launch(args) ;
    }

}

// Event Handler class
class CmdHandler implements EventHandler<ActionEvent>
{
    // These class variables reference
    // data in the calling environment.
    private HBox view;
    private TextField cmdResultTextField;
    private BinarySearchTree binTree;
    private DisplayableBtree displayBinTree;

    // Constructor stores references to data in
    // the calling environment.
    public CmdHandler(HBox tView, TextField cmdResTf,
                      BinarySearchTree tree,
                      DisplayableBtree dTree)
    {
        view = tView;
        cmdResultTextField = cmdResTf;
        binTree = tree;
        displayBinTree = dTree;
    }

    public void handle(ActionEvent evt)
    {
        // Get the command from the event target.
        TextField cmdTf = (TextField)evt.getTarget();
        String cmdStr =  cmdTf.getText();

        // Use a scanner to read the command and its parts.
        Scanner sc = new Scanner(cmdStr);
        String cmd = sc.next();
        switch (cmd)
        {
            case "add":
            {
                // Perform the operation on the tree.
                int value = sc.nextInt();
                binTree.add(value);

                // Replace the old view of the tree.
                view.getChildren().clear();
                view.getChildren().add(displayBinTree.getView());
                cmdResultTextField.setText(" ");
            }
            return;
            case "isempty":
            {
                String resultText
                        = "Tree empty? " + binTree.isEmpty();
                cmdResultTextField.setText(resultText);
            }
            return;
            case "contains":
            {
                int value = sc.nextInt();
                String resultText = "Contains " + value + "?: "
                        + binTree.contains(value);
                cmdResultTextField.setText(resultText);
            }
            return;
            case "remove":
            {
                int value = sc.nextInt();
                binTree.remove(value);
```

```

                 // Replace the old view of the tree.
                view.getChildren().clear();
                view.getChildren().add(displayBinTree.getView());
                cmdResultTextField.setText(" ");
            }
            return;
        }
    }
}
```

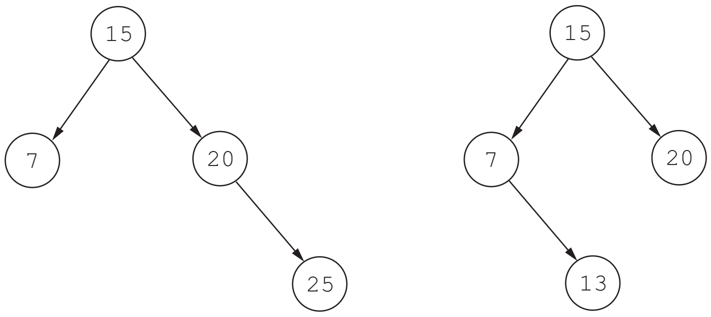
## 21.3 AVL Trees

**CONCEPT: Operations on binary search trees are more efficient when the tree is balanced. An AVL tree is a binary search tree that satisfies a balance condition at each one of its nodes.**

Because most operations on binary search trees trace out a path from the root to some node, the amount of work they do is proportional to the length of the path. Therefore it is desirable to have binary search trees that are “shallow” rather than “deep.” A binary tree can be kept shallow by imposing conditions on the height of its subtrees. The *height* of a binary tree is the length of the longest path from the root to a leaf. (The length of a path is the number of edges, or links, in the path: it is one less than the number of nodes in the path.) A binary tree with one node has height 0, and by convention, the empty binary tree has height −1.

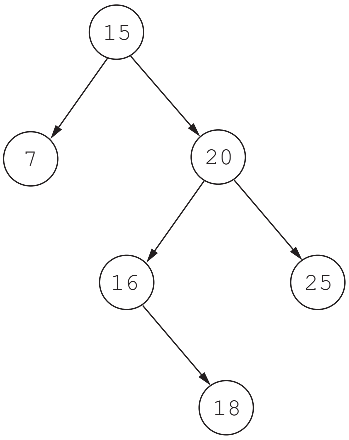
An *AVL tree* is a binary search tree in which the heights of the left and right subtree of every node differ by at most 1. Figure 21-13 shows two examples of AVL trees.

**Figure 21-13** Examples of AVL trees



The tree of Figure 21-14 fails to be an AVL tree because the right subtree of the root has height 2 whereas the left subtree has height 0.

**Figure 21-14** An example of a non-AVL tree



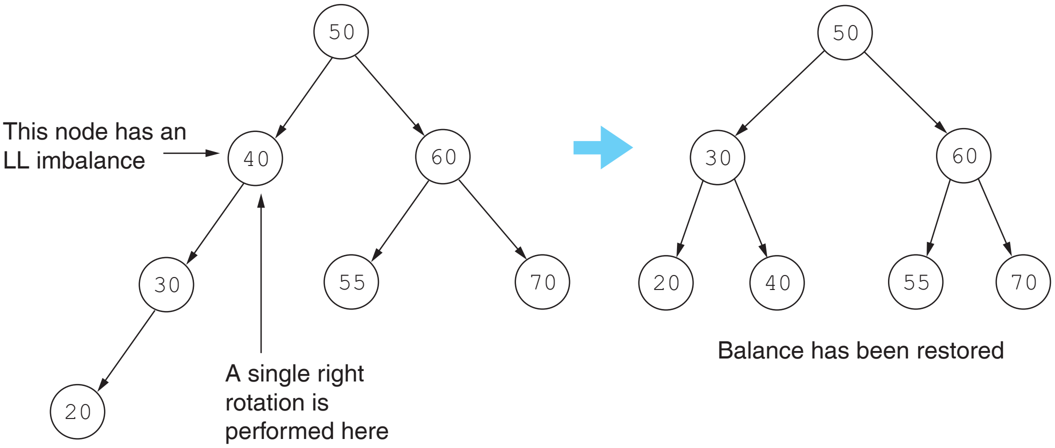
## Adding New Elements to AVL Trees

AVL trees support the same operations as regular binary search trees: addition of new elements, removal, and search. To add a new element to an AVL tree, we simply add the new element using the same algorithm used to add elements to a regular binary search tree. It is possible, however, that the addition may upset the AVL balance condition, so we must check for this and rebalance the tree as necessary. We only need to check the balance condition at nodes that lie on the path between the newly inserted node and the root.

There are four types of imbalance that can result from adding a new node to an existing AVL tree:

- An *LL imbalance* occurs at a node *N* with a left child *K* when *N* and *K* are both left-heavy: that is, the height of the left subtree of *N* is greater than the height of the right subtree of *N*; and the height of the left subtree of *K* is greater than the height of the right subtree of *K*. An LL imbalance is fixed by performing a *single right rotation* at *N*: this is illustrated in Figure 21-15.
- An *RR imbalance* occurs at a node *N* with a right child *M* when *N* and *M* are both right-heavy. This situation is the mirror image of an LL imbalance: it is fixed by performing a *single left rotation* at *N*.
- An *LR imbalance* occurs at a node *N* with a left child *K* when *N* is left-heavy and *K* is right-heavy. An LR imbalance is fixed by performing a *double LR rotation* at *N*. This is illustrated in Figure 21-16.
- An *RL imbalance* occurs at a node *N* with a right child *M* when *N* is right-heavy and *M* is left-heavy. This situation is a mirror image of an LR imbalance, and can be fixed by performing a *double RL rotation* at *M*.

**Figure 21-15** A single right rotation is performed at a node with an LL imbalance

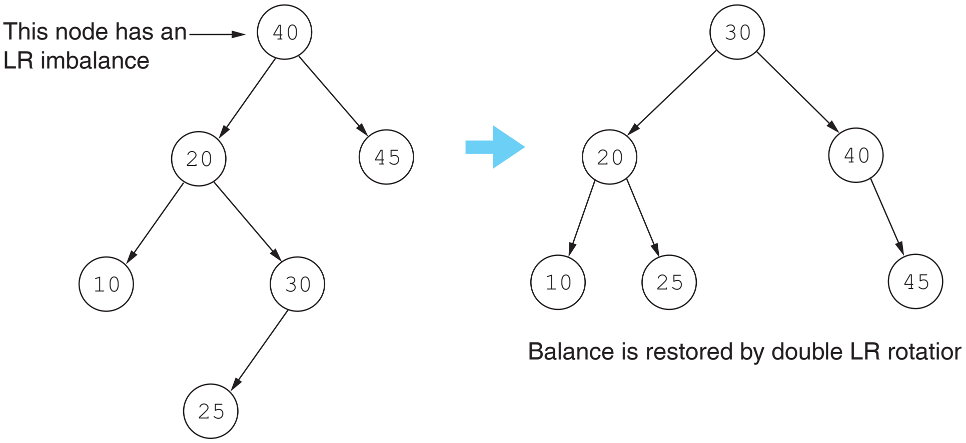


This node has an LL imbalance

A single right rotation is performed here

Balance has been restored

**Figure 21-16** A double LR rotation is performed at a node with an LR imbalance

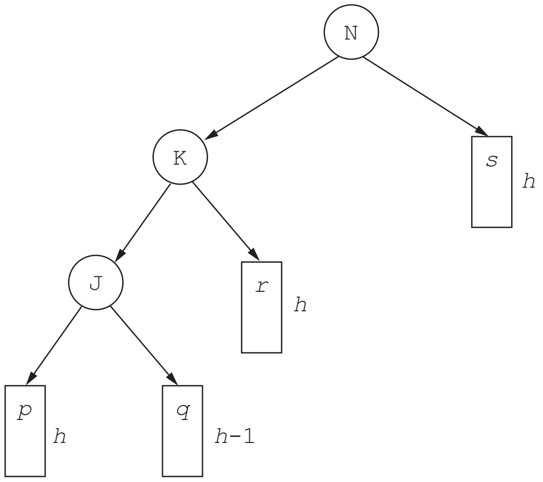


This node has an LR imbalance

Balance is restored by double LR rotatior

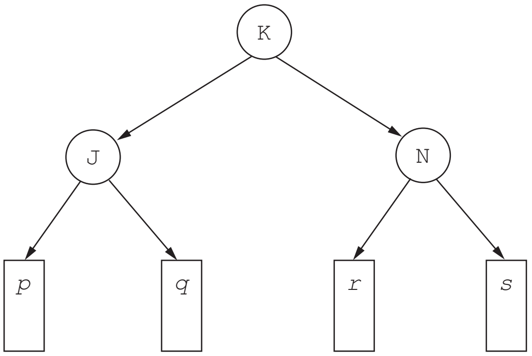
Consider now a general strategy for removing an LL imbalance that occurs after a new node has been added to an AVL tree. Let $N$ be the node with the LL imbalance, let $K$ be the left child of $N$, and let $J$ be the left child of $K$. As shown in Figure 21-17, let $p$, $q$, $r$, and $s$ be respectively the left subtree of $J$, the right subtree of $J$, the right subtree of $K$, and the right subtree of $N$. Letting $h$ be the height of $s$, we deduce that the height of the left subtree of $N$ must be $h + 2$. We can further deduce that the height of $r$ must be $h$, and that one of $p$ and $q$ must have height $h$ while the other has height $h - 1$. Let us assume that $p$ has height $h$ and $q$ has height $h - 1$.

**Figure 21-17** A node with an LL imbalance



In general, this situation occurs when a new node has been added to the subtree *p*, and the imbalance is discovered at *N* as we are tracing our way back to the root. At this point, we fix the imbalance by performing a single right rotation at *N*, that is, we rearrange the parent–child relationships among the nodes and subtrees shown to yield the tree shown in Figure 21-18.

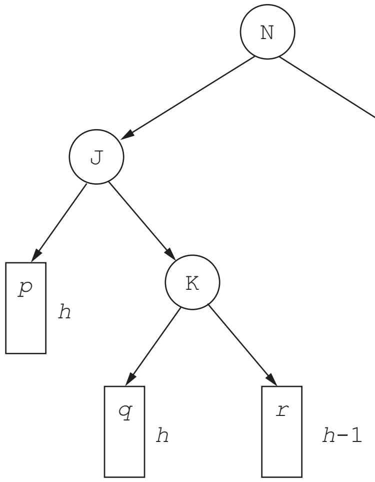
**Figure 21-18** Effect of a single right rotation



Notice that the rearranged tree is still a binary search tree, and moreover, it now obeys the AVL balance condition at every node. An RR imbalance can similarly be fixed using a single left rotation.

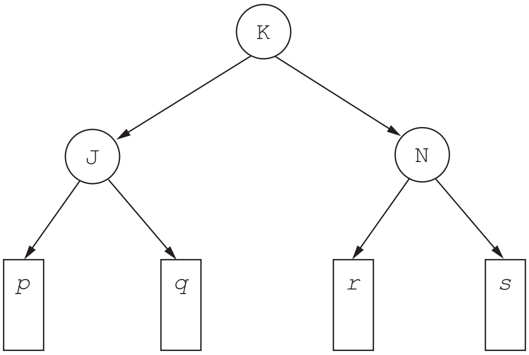
Consider now how to remove an LR imbalance at a node *N*. Any such node *N* will have a left child *J*, which will in turn have a right child *K*. Let *p*, *q*, *r*, and *s* represent the left and right subtrees of these nodes as shown in Figure 21-19. Assuming that *s* has height *h*, we deduce that the subtree rooted at *J* must have height $h + 2$. We can further deduce that the height of *p* is *h*. Because the subtree rooted at *K* has height $h + 1$, at least one of the subtrees of *K*, say *q*, will have height *h*. The other subtree, *r*, must have height $h - 1$ (*r* cannot have a height less than $h - 1$ because then the imbalance would be found at *K* rather than at *N*).

**Figure 21-19** A node with an LR imbalance



The LR imbalance shown in Figure 21-19 is removed by executing a double LR rotation, which rearranges the tree into the form shown in Figure 21-20.

**Figure 21-20** Effect of a double LR rotation on the tree of Figure 21-19



The reconfigured tree is now an AVL tree. An RL imbalance can similarly be removed by performing a double RL rotation.

## Implementation of AVL Trees

An AVL node is a binary tree node with a height field. Our implementation of an AVL tree is based on the `AVLNode` class, which is a subclass of Node. The `AVLNode` class inherits the

```
int value;
Node left;
Node right;
```

fields from its superclass and adds an integer field to keep track of the height of the node in the AVL tree. The `Node` references `left` and `right` work fine when being assigned to, or when being used to access the fields inherited from `Node`, but require casting to access the height field:

```
int h =  ((AVLNode)left).height;
```

To simplify the use of these references, we include in the `AVLNode` class convenience methods that apply the cast. You can see these methods in Lines 12 and 13 of Code Listing 21-10.

**Code Listing 21-10** (`AVLNode.java`)

```
 1 /**
 2   AVLNode extends Node from which it inherits
 3   the value field, and also the left and right links.
 4 */
 5
 6 class AVLNode extends Node
 7 {
 8     int height;
 9
10     // These methods cast the inherited NODE links
11     // AVLNode links.
12     AVLNode getLeft(){ return (AVLNode) left; }
13     AVLNode getRight(){ return (AVLNode) right; }
14
15     public AVLNode(int value)
16     {
17         // Call the other (sister) constructor.
18         this(value, null, null);
19     }
20
21     public AVLNode(int val, AVLNode left1, AVLNode right1)
22     {
23         // Pass the parameters to the superclass constructor.
24         super(val, left1, right1);
25         height = 0;
26     }
27
28     /**
29       The resetHeight methods recomputes height if the
30       left or right subtrees have changed.
31     */
32     void resetHeight()
33     {
34         int leftHeight = -1;
35         int rightHeight = -1;
36         if (left != null)
```

```
        {
            leftHeight = getLeft().height;
        }
        if (right != null)
        {
            rightHeight = getRight().height;
        }
        height = 1 + Math.max(leftHeight, rightHeight);
    }
}
```

The implementation of AVL trees is similar to that of binary search trees, except that the add and remove methods must be modified to ensure that the tree stays balanced. Code Listing 21-11 shows an AVL class `AVLTree` with the add method implemented. The class extends the `BinaryTree` class of Code Listing 21-6, from which it inherits the `root` field. Although declared as being of type `Node`, the `root` field will point to `AVLNode` objects. Accordingly, the `AVLTree` class has a `getRoot()` method that retrieves `root` and automatically applies a cast to `AVLNode`. The `getRoot()` method appears in Lines 8–11 of Code Listing 21-11.

The `add(bTree,x)` method is a straightforward implementation of the strategy already discussed. If `bTree` is empty, add creates and returns a new AVL tree with a single node. Otherwise, the method recursively adds the new value to one of the subtrees, recomputes the height of the subtrees, and checks the balance condition at `bTree`. If the `bTree` is out of balance, add calls the balance method on it.

The balance method determines the type of imbalance existing at the node passed to it and calls one of four methods to perform the appropriate rotation. The methods that perform the rotations are named `llBalance`, `lrBalance`, `rlBalance`, and `rrBalance` after the type of imbalance they correct.

**Code Listing 21-11** (`AVLTree.java`)

```
 1 /**
 2   This class implements AVLTrees.
 3 */
 4 public class AVLTree extends BinaryTree
 5 {
 6     // Convenience method casts the inherited root
 7     // from Node to AVLNode.
 8     private AVLNode getRoot()
 9     {
10         return (AVLNode) root;
11     }
12
13     /**
14       The getHeight method computes the height of an AVL tree.
15       @param tree An AVL tree.
16       @return The height of the AVL tree.
17     */
```

```
    static int getHeight(AVLNode tree)
    {
        if (tree == null)
            return -1;
        else
            return tree.height;
    }

    /**
      The add method adds a value to this AVL tree.
      @param x The value to add.
      @return true.
    */
    public boolean add(int x)
    {
        root = add((AVLNode) root, x);
        return true;
    }

    /**
      The add method adds a value to an AVL tree.
      @return The root of the augmented AVL tree.
    */
    private AVLNode add(AVLNode bTree, int x)
    {
        if (bTree == null)
        {
            return new AVLNode(x);
        }
        if (x < bTree.value)
        {
            bTree.left = add(bTree.getLeft(), x);
        } else
        {
            bTree.right = add(bTree.getRight(), x);
        }

        // Compute heights of the left and right subtrees
        // and rebalance the tree if needed.
        int leftHeight = getHeight(bTree.getLeft());
        int rightHeight = getHeight(bTree.getRight());
        if (Math.abs(leftHeight - rightHeight) == 2)
        {
            return balance(bTree);
        } else
        {
            bTree.resetHeight();
            return bTree;
        }
    }

    /**
      The balance method rebalances an AVL tree.
      @param bTree The AVL tree needing to be balanced.
      @return The balanced AVL tree.
    */
    private AVLNode balance(AVLNode bTree)
    {
        int rHeight = getHeight(bTree.getRight());
        int lHeight = getHeight(bTree.getLeft());

        if (rHeight > lHeight)
        {
            AVLNode rightChild = bTree.getRight();
            int rrHeight = getHeight(rightChild.getRight());
            int rlHeight = getHeight(rightChild.getLeft());
            if (rrHeight > rlHeight)
            {
                return rrBalance(bTree);
            } else
            {
                return rlBalance(bTree);
            }
        } else
        {
            AVLNode leftChild = bTree.getLeft();
            int llHeight = getHeight(leftChild.getLeft());
            int lrHeight = getHeight(leftChild.getRight());
            if (llHeight > lrHeight)
            {
                return llBalance(bTree);
            } else
            {
                return lrBalance(bTree);
            }
        }
    }

    /**
      The rrBalance method corrects an RR imbalance.
      @param bTree The AVL tree wih an RR imbalance.
      @return The balanced AVL tree.
    */
    private AVLNode rrBalance(AVLNode bTree)
    {
        AVLNode rightChild = bTree.getRight();
        AVLNode rightLeftChild = rightChild.getLeft();
        rightChild.left = bTree;
        bTree.right = rightLeftChild;
        bTree.resetHeight();
        rightChild.resetHeight();
        return rightChild;
    }

    /**
      The rlBalance method corrects an RL imbalance.
      @parame bTree The AVL tree with an RL imbalance.
      @return The balanced AVL tree.
    */
    private AVLNode rlBalance(AVLNode bTree)
    {
        AVLNode root = bTree;
        AVLNode rNode = root.getRight();
        AVLNode rlNode = rNode.getLeft();
        AVLNode rlrTree = rlNode.getRight();
        AVLNode rllTree = rlNode.getLeft();

        // Build the restructured tree.
        rNode.left = rlrTree;
        root.right = rllTree;
        rlNode.left = root;
        rlNode.right = rNode;

        // Adjust heights
        rNode.resetHeight();
        root.resetHeight();
        rlNode.resetHeight();

        return rlNode;
    }

    /**
      The llBalance method corrects an LL imbalance.
      @param bTree The AVL tree with an LL imbalance.
      @return The balanced AVL tree.
    */
    private AVLNode llBalance(AVLNode bTree)
    {
        AVLNode leftChild = bTree.getLeft();
        AVLNode lrTree = leftChild.getRight();
        leftChild.right = bTree;
        bTree.left = lrTree;
        bTree.resetHeight();
        leftChild.resetHeight();
        return leftChild;
    }

    /**
      The lrBalance method corrects an LR imbalance.
      @param bTree The AVL tree with an LR imbalance.
      @return The balanced AVL tree.
    */
    private AVLNode lrBalance(AVLNode bTree)
    {
        AVLNode root = bTree;
        AVLNode lNode = root.getLeft();
        AVLNode lrNode = lNode.getRight();
        AVLNode lrlTree = lrNode.getLeft();
        AVLNode lrrTree = lrNode.getRight();

        // Build the restructured tree.
        lNode.right = lrlTree;
        root.left = lrrTree;
        lrNode.left = lNode;
        lrNode.right = root;

        // Adjust heights.
        lNode.resetHeight();
        root.resetHeight();
        lrNode.resetHeight();

        return lrNode;
    }
}

```

Code Listing 21-12 implements the graphical interface to the `AVLTree` class. The program is a simple modification of the one in Code Listing 21-9.

**Code Listing 21-12** (`GUIAVLTreeDemo.java`)

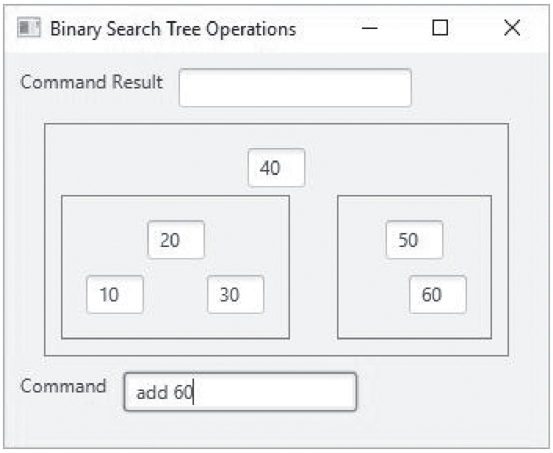
```
import java.util.Scanner;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

```

```
/**
   This class is a JavaFX interface for testing the
   the binary search tree class.
*/

public class GUIAVLTreeDemo extends Application
{
    @Override
    public void start(Stage stage) throws Exception
    {
       // Set up the binary tree to manipulate

       AVLTree avlTree = new AVLTree();

       // This binary tree wraps the real binary tree
       // so it can be displayed.
       DisplayableBtree
           displayBinTree = new DisplayableBtree(avlTree);

       // Displays the result of the user command.
       TextField cmdResultTextField = new TextField();
       cmdResultTextField.setEditable(false);

       // Used by the user to enter a command.
       TextField cmdTextField = new TextField();

       // HBox1 holds cmdResultTextField and its label.
       HBox hBox1 = new HBox(10);
       Label cmdResultLabel = new Label("Command Result");
       hBox1.getChildren().addAll(cmdResultLabel, cmdResultTextField);

       // Place holder for the binary tree view.
       HBox bTreeView = new HBox();
       bTreeView.setAlignment(Pos.CENTER);

       // HBox2 holds the cmdTextField and its label
       HBox hBox2 = new HBox(10);
       Label cmdLabel = new Label("Command");
       hBox2.getChildren().addAll(cmdLabel, cmdTextField);

       // Put the three HBox objects in a VBox
       VBox outerPane = new VBox(10);
       outerPane.setPadding(new Insets(10));
       outerPane.getChildren().addAll(hBox1, bTreeView, hBox2);

       // Command handler needs to access to the bTreeView
       // so it can set the binary tree view, and to the
       // cmdResultTextField so it can set the command result.
       // Also needs access to the binary tree and its displayable
       // twin.
```

```
        EventHandler<ActionEvent>
            handler = new CmdHandler(bTreeView, cmdResultTextField,
                                     avlTree, displayBinTree);
        cmdTextField.setOnAction(handler);

        // Set up the stage
        stage.setTitle("Binary Search Tree Operations");
        stage.setScene(new Scene(outerPane));
        stage.show();
    }

    public static void main(String [] args)
    {
         launch(args) ;
    }

}

// Event Handler class
class CmdHandler implements EventHandler<ActionEvent>
{
    // These class variables reference
    // data in the calling environment.
    private HBox view;
    private TextField cmdResultTextField;
    private AVLTree binTree;
    private DisplayableBtree displayBinTree;

    // Constructor stores references to data in
    // the calling environment.
    public CmdHandler(HBox tView, TextField cmdResTf,
                      AVLTree tree,
                      DisplayableBtree dTree)
    {
        view = tView;
        cmdResultTextField = cmdResTf;
        binTree = tree;
        displayBinTree = dTree;
    }

    public void handle(ActionEvent evt)
    {
        // Get the command from the event target.
        TextField cmdTf = (TextField)evt.getTarget();
        String cmdStr =  cmdTf.getText();

        // Use a scanner to read the command and its parts.
        Scanner sc = new Scanner(cmdStr);
        String cmd = sc.next();
        switch (cmd)
        {
            case "add":
            {
                // Perform the operation on the tree.
                int value = sc.nextInt();
                binTree.add(value);

                // Replace the old view of the tree.
                view.getChildren().clear();
                view.getChildren().add(displayBinTree.getView());
                cmdResultTextField.setText(" ");
            }
            return;
        }
    }
}
```

A screenshot of this user interface after the user has issued commands to add the first six multiples of 10 to an initially empty AVL tree is shown in Figure 21-21.

**Figure 21-21** Sample interaction with AVL program (Mark Schrier. Copyright Oracle, Inc.)



add 60

## 21.4 Priority Queues

**CONCEPT: A priority queue is a collection that stores elements according to some order. Priority queues support addition and removal of elements, and a removal always returns the least element in the collection.**

An *order* on a set of values is a way of establishing, for any two elements $x$ and $y$, whether $x$ is "less than" $y$ or $y$ is "less than" $x$. Many sets have a *natural order*: for example, the set of integers is ordered by the usual "less than" relation $<$, that is, in the natural order for the integers, $x$ is "less than" $y$ if and only if $x < y$. Likewise, in the natural order for strings, a string $s$ is "less than" string $t$ if $s$ comes before $t$ in alphabetical order. In an ordered set, an element that is "less than" every other element is called the *least* element, or the *minimum*. All orders referred to in this chapter will be assumed to be *total* (or *linear*), meaning that for any two elements $x$ and $y$, exactly one of the following conditions must hold:

1. $x$ is "less than" $y$
2. $x$ is equal to $y$
3. $y$ is "less than" $y$

As an example of an order that is not natural, we can stipulate, for any two numbers $x$ and $y$, that $x$ is "less than" $y$ if $x > y$. In this case, for a set such as $\{1, 2, 3, 4\}$, the defined order turns the set into the sequence $\{4, 3, 2, 1\}$, and makes 4 the "least" element of the set.

The natural order of primitive types in Java is specified by the built-in comparison operator $<$. A class type such as `Integer` or `String` specifies its natural order by implementing the `Comparable` interface.

A *priority queue* is a collection that stores elements and provides access to those elements according to some order. A remove operation executed on a priority queue always returns a minimum element. The queue element that will be returned by the next call to remove is said to be at the *head of the queue*.

The main operations supported by a priority queue when regarded as an abstract data structure are as follows:

- *add*($x$): Adds $x$ to the priority queue.
- *removeMin*( ): Removes and returns the element at the head of the queue; this will be an element that is less than, or equal to, any other element in the queue.

The Java Collection Framework provides an implementation of a priority queue in a class called `PriorityQueue`. It provides more methods than the two we have listed, and some of the method names are different. Most of its methods are listed in Table 21-1.

**Table 21-1 `PriorityQueue` class methods**

| | |
|---|---|
| `PriorityQueue<E>()` | Creates a queue that stores items of class type `E` and orders them according to their natural ordering (using the Comparable interface as implemented by `E`). |
| `PriorityQueue<E>(int capacity, Comparator<? Super E> comp)` | Creates a queue that stores items of class `E` and orders them by the specified comparator. |
| `boolean add(E item)` | Adds `item` to the priority queue and returns `true`. |
| `E poll()` | Removes and returns a minimum element from the priority queue. This method returns `null` if the queue is empty. |
| `E peek()` | Returns the item that is currently at the head of the queue, but does not remove it from the queue. If the queue is empty, `peek` returns `null`. |
| `int size()` | Returns the number of items currently stored in this priority queue. |

## Applications of Priority Queues

Priority queues are useful whenever items have to be processed according to some priority. For example, a personal information manager application might keep a list of tasks to be completed in a priority queue ordered by deadline, with the minimum task being the one with the closest deadline. Priority queues are also used by computer operating systems to schedule tasks for execution, and in computer networks to schedule data packets for transmission.

If you store a set of items in a priority queue and then retrieve them, they will come out in sorted order. This is the basis of a very efficient sorting algorithm named *Heapsort*. Code Listing 21-13 illustrates the use of a priority queue to sort an array.

**Code Listing 21-13** `(Heapsort.java)`

```
import java.util.*;

/**
   This program demonstrates the Heapsort algorithm.
*/

public class Heapsort
{
   public static void main(String [] args)
   {
      // Create and display an array of random integers
      Random randy = new Random();
      int [ ] arr = new int[10];
      System.out.println("Here is the array to be sorted:");
      for (int k = 0; k < arr.length; k++)
      {
         arr[k] = randy.nextInt(100);
         System.out.print(arr[k] + "  ");
      }

      // Create a priority queue of integers
      // and use it to sort the array
      PriorityQueue<Integer> pQueue =
         new PriorityQueue<Integer>();
      for (int x : arr) pQueue.add(x);
      for (int k = 0; k < arr.length; k++)
      arr[k] = pQueue.poll();

      // Print the array
      System.out.println("\nHere is the sorted array:");
      for (int x : arr) System.out.print(x + "  ");
   }
}
```

**Program Output**

```
Here is the array to be sorted:
13  81  78  93  94  60  2  32  88  58
Here is the sorted array:
2  13  32   58  60  78  81  88  93  94
```

## Using Comparators with Priority Queues

The no-argument constructor of the `PriorityQueue` class used in Code Listing 21-13 orders items according to their natural ordering. This is the order specified by the < operator if the type of items is primitive, or specified by the `Comparable` interface for class types that implement it. If we need to use a different order, we must use the second constructor shown in Table 21-1. That constructor takes two parameters: an integer specifying the capacity of the queue to be created, and a `Comparator` object. `Comparator` is a generic interface defined as follows:

```
interface Comparator<T>
{
   int compare (T x, T y);
   boolean equals(Object o);
}
```

A class implementing this interface provides a definition of `compare` that returns a negative integer if *x* should be regarded as less than *y*, a positive integer if *x* should be regarded as greater than *y*, and 0 if *x* is equal to *y*. The `equals` method of the `Comparator` interface exists to allow two `Comparator` objects to be checked for equality, and does not have to be implemented if the default implementation inherited from `Object` is adequate.

As an example, let us look at a version of Heapsort that sorts integers in alphabetical order of their string representation. This is done by creating a `PriorityQueue` object that uses a `Comparator` object defined from the following class:

```
class AlphaOrder implements Comparator<Integer>
{
   public int compare(Integer x, Integer y)
   {
      return x.toString().compareTo(y.toString());
   }
}
```

The `AlphaOrder` class is defined inside `Alphaheapsort` as a static inner class. `AlphaOrder` is made static because it will be instantiated by the `main` method, which is itself static. In Java, static methods cannot instantiate non-static inner classes.

The following code fragment creates a priority queue that uses `AlphaOrder` as a comparator, and the program in Code Listing 21-14 demonstrates its use:

```
AlphaOrder comp = new AlphaOrder();
PriorityQueue<Integer>
        pQueue = new PriorityQueue<Integer>(arr.length, comp);
```

**Code Listing 21-14** `(Alphaheapsort.java)`

```java
import java.util.*;

/**
   This program shows the use of a PriorityQueue with
   a custom Comparator object.
*/

public class Alphaheapsort
{
   /**
      Specialized Comparator.
   */

   static class AlphaOrder implements Comparator<Integer>
   {
      public int compare(Integer x, Integer y)
      {
         return x.toString().compareTo(y.toString());
      }
   }

   public static void main(String [] args)
   {
      // Create and display an array of random integers
      Random randy = new Random();
      int [ ] arr = new int[10];
      String str = "Here is the original list of numbers:";
      System.out.println(str);
      for (int k = 0; k < arr.length; k++)
      {
         arr[k] = randy.nextInt(10000);
         System.out.print(arr[k] + "  ");
      }

      // Create a priority queue of integers
      // and use it to sort the array
      AlphaOrder c = new AlphaOrder();
      PriorityQueue<Integer>
         pQueue = new PriorityQueue<Integer>(arr.length, c);
      for (int x : arr)
         pQueue.add(x);
      for (int k = 0; k < arr.length; k++)
         arr[k] = pQueue.poll();

      // Print the array
      System.out.println("\nHere are the numbers sorted " +
                              "in alphabetical order:");
         for (int x : arr)
            System.out.print(x + "  ");
      }
   }
```

**Program Output**

```
Here is the original list of numbers:
1065  8433  743  4962  9954  31  1815  4135  489  3094
Here are the numbers sorted alphabetical in order:
1065  1815  3094  31  4135  489  4962  743  8433  9954
```

## Analysis of Heapsort

We can analyze the efficiency of Heapsort by counting the number of comparisons of array elements that it causes the priority queue to perform. It turns out that a properly implemented priority queue holding $n$ elements requires at most $\log n$ comparisons to add or remove an item. Given an array of $n$ elements, we can add all of them to a priority queue using $n\log n$ comparisons, and then remove all of them using another $n\log n$ comparisons, for a total of $2n\log n$ comparisons. In the language of Chapter 16, Heapsort is a $O(n\log n)$ algorithm. In practice, an $n\log n$ algorithm is a lot faster than a $O(n^2)$ algorithm such as Selectionsort or Bubblesort. It can be shown that no algorithm that sorts by comparing items can sort an array of $n$ items in less than $n\log n$ comparisons. This means that, within a constant factor, the performance of Heapsort is the best possible.

## Implementation of Priority Queues

Our goal in implementing a priority queue is to find a data structure that allows each of the two main priority queue operations, *add* and *removeMin*, to be performed with no more than $\log n$ comparisons when the priority queue has $n$ items. Two possibilities immediately come to mind:

1. We can use an unsorted linked list to hold the $n$ items. This will allow us to add a new element to the list quickly, but will force us to perform $n - 1$ comparisons to find and remove the minimum element.
2. We can use a linked list sorted in increasing order to hold the $n$ items. This allows us to quickly find and remove the minimum, but will force us, in the worst case, to perform $n$ comparisons while looking for the right place to insert the new item.

Neither of these achieves the desired $O(\log n)$ complexity for both operations, so we must look for another approach.

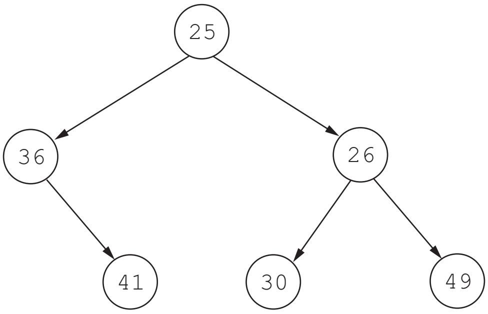
## Binary Trees with the Heap Order Property

A third possibility is to use a binary tree to hold the elements of the queue. Consider a binary tree $T$ that satisfies the following condition, which we will dub *the heap order property*:

At every node $N$ of $T$, the value stored in $N$ is greater than the value stored in the parent of $N$.

Intuitively, a binary tree with this property is a tree in which every path from the root to a leaf is sorted in increasing order. Figure 21-22 shows an example of such a tree.

**Figure 21-22** A binary tree with the heap order property



We need to impose a condition on such a tree that will keep it shallow; that is, we want the length of the longest path through the tree to be approximately log $n$ rather than $n$. This will ensure that an addition operation that has to traverse a path from the root to a leaf can be performed in log $n$ time. We need to introduce some terminology before we can proceed.

Let $T$ be a binary tree. The *level* of a node $N$ in $T$ is the length of a path from the root to $N$. For example, the level of the root is 0, and the level of a child of a root is 1. The *depth* of $T$ is the maximum level of a node in $T$; that is, it is the length of the longest path from the root to a leaf. A binary tree with depth $d$ has $d + 1$ levels, because it has a level for each value of $l$ in the range $0 \leq l \leq d$. We are looking for a family of trees whose depth is at most log $n$ when the tree has $n$ nodes.

## Complete Binary Trees and Heaps

A binary tree with depth $d$ is *complete* if it satisfies two conditions:

- $T$ has $2^l$ nodes for all levels $0 \leq l \leq d - 1$; that is, each level other than the last has the maximum number of nodes possible
- All leaf nodes at level $d$ are as far to the left as possible

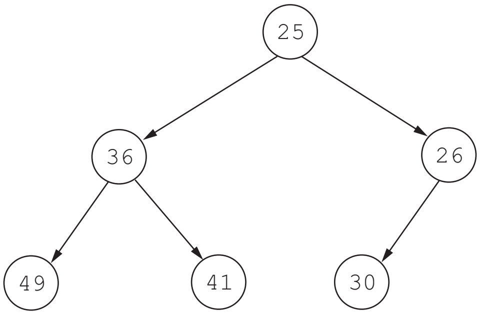
Basically, a complete binary tree is one that you get when you build a binary tree by starting with the root and adding nodes level by level, following these rules:

R1. No node gets a child until each node at the previous level has two children.
R2. No node gets a child unless its left sibling already has two children.
R3. No node gets a right child unless it already has a left child.

The first rule ensures that there are no “holes” in the first $d$ levels of the tree. The next two rules ensure that there are no “holes” as you move from left to right through the last level.

The tree of Figure 21-22 is not complete because the node that contains 36 has a right child but no left child. Figure 21-23 shows an example of a complete binary tree.

**Figure 21-23** A complete binary tree that is also a heap



A complete binary tree that stores values from an ordered set is called a *heap* if it satisfies the heap order property.

## The Depth of Complete Binary Trees

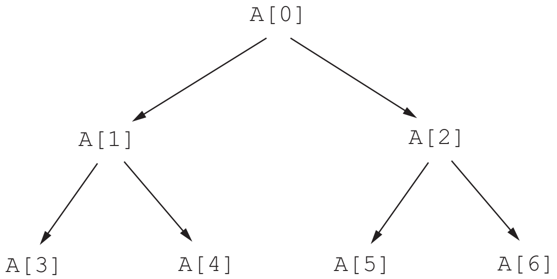
A complete binary tree $T$ with $n$ nodes is shallow in the sense that it cannot have depth much greater than log $n$. If we disregard the last level, what remains of $T$ is perfectly balanced in the sense that at each node, there are just as many nodes in the left subtree as in the right subtree. This means that if we start at the root and descend along any path, we retain less than half of the remaining nodes each time we pass to a new (numerically higher) level. This shows that the length of any path can be at most $\log n + 1$. We state this as follows:

> A complete binary tree with $n$ nodes has depth $d \leq \log n + 1$.

## Storing a Complete Binary Tree in an Array

The fact that a complete binary tree has no "holes" has an interesting consequence. If you build the binary tree using rules R1 to R3 and store each node in an array $A$ as it is added to the tree, you get the pattern shown in Figure 21-24.

**Figure 21-24** Storing a complete binary tree in an array



This pattern has the following characteristics:

- The root of the tree is $A[0]$
- The parent of a node $A[k]$ that is not the root is $A[(k - 1)/2]$

- The left child and right children of a node A[$k$] are $A[2k + 1]$ and $A[2k + 2]$, respectively
- The rightmost leaf in the last level (last node in the tree) is at $A[n - 1]$
- A node $A[k]$ is a leaf if $2k + 1 \geq n$, it has a left child if $2k + 1 < n$, and has a right child if $2k + 2 < n$

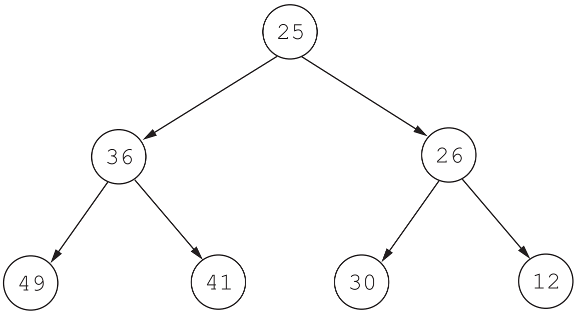
These observations allow us to use either an array or an array list to store a complete binary tree. In particular, to build a heap of integers, we can use an array list such as the following:

```
ArrayList<Integer> arrayHeap;
```

## Adding an Item to a Heap

An addition of a new item to a heap must maintain the structure of a complete binary tree as well as the heap order property. We can preserve the structure of the tree by adding the new node according to rules R1, R2, and R3. This means that the new node will be added as a leaf at the last level of the tree, and will be as far to the left as possible within that level. For example, adding 12 to the tree of Figure 21-23 yields the tree of Figure 21-25.

**Figure 21-25** Adding a new value may violate the heap order property



In terms of our array list representation, this means that the new value is added to the end of the array list as follows:

```
arrayHeap.add(x);
```

Once we have added the new node, say $X$, we restore the heap order property by traversing the path from the point of insertion back to the root. As we go, we compare the value of $X$ to that of its parent and swap the two nodes if the value in the parent is greater. Then we climb up the tree to the new position of $X$ and repeat. We stop when the $X$ becomes the root, or when it gets to a place where its value is greater than that of its parent. When we do this, we say we are *sifting up* the new node. The code for adding a value to our heap is embodied in the following method:

```
boolean add(int x)
{
   // Add x at the end of the array list
   arrayHeap.add(x);
   // Sift up
```

```
      siftUp();
      return true;
   }
```

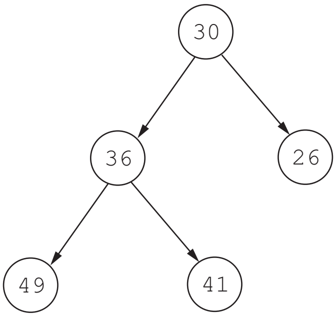
The sifting up procedure is performed by the following method:

```
   private void siftUp()
   {
      int p = arrayHeap.size()-1;  // Position to sift up
      while (p != 0)
      {
         int parent = (p-1) / 2;    // Index of parent
         if (valueAt(p) >= valueAt(parent))
         {
            // We are done
            return;
         }
         else
         {
            // Do a swap
            Integer temp = arrayHeap.get(parent);
            arrayHeap.set(parent, arrayHeap.get(p));
            arrayHeap.set(p, temp);
            // Move up
            p = parent;
         }
      }
   }
```

## Removing the Minimum Element

To remove an element from a heap, we delete the root node and replace it with the node in the last place of the queue, that is, the rightmost leaf at the deepest level of the tree. If we do this to the heap of Figure 21-23, we get the complete binary tree shown in Figure 21-26.

**Figure 21-26** The heap of Figure 21-23 after replacing the root with the deepest rightmost leaf



In general, the tree will violate the heap order property when the root is replaced in this manner, and we must perform a *sift down* operation to remake the heap. Say that *X* is the value that is currently stored in the root and needs to be sifted down. We start at the root and make our way down some branch of the tree. If we get to a place where *X* is smaller than its children, or is a leaf, we stop the procedure. Otherwise, we swap *X* with its smallest child, and then follow *X* down to the next level. In this way, we sift *X* down the tree until we get it to where it belongs. At that point, the heap order property is restored and we have remade the tree into a heap.

In terms of our array implementation, we must first save the value at position 0, and then remove the value at the end of the array list and store it at position 0. The `removeMin` method is as follows:

```
public int removeMin()
{
  if (isEmpty())
    throw new RuntimeException("Priority Queue is empty.");
  else
  {
    int val = arrayHeap.get(0);

    // Replace root by last leaf
    arrayHeap.set(0, arrayHeap.get(arrayHeap.size()-1));

    // Remove the last leaf
    arrayHeap.remove(arrayHeap.size()-1);
    siftDown();
    return val;
  }
}
```

You can see the `siftDown` method, together with the methods for `add` and `removeMin`, in Code Listing 21-15.

**Code Listing 21-15** (`Arrayheap.java`)

```
import java.util.*;

/**
   This class is an array-based heap
*/

public class ArrayHeap
{
   private ArrayList<Integer> arrayHeap
                  = new ArrayList<>();

   /**
      The siftUp method sifts up the element
      at arrayHeap[size()-1].
   */
   private void siftUp()
   {
      int p = arrayHeap.size()-1;  // Position to sift up
      while (p != 0)
      {
         int parent = (p-1) / 2;
         if (valueAt(p) >= valueAt(parent))
         {
            // We are done
            break;
         }
         else
         {
            // Do a swap
            Integer temp = arrayHeap.get(parent);
            arrayHeap.set(parent, arrayHeap.get(p));
            arrayHeap.set(p, temp);

            // Move up
            p = parent;
         }
      }
   }

   /**
      The siftDown method sifts down the
      element at arrayHeap[0].
   */

   private void siftDown()
   {
      int p = 0; // Position to sift down
      int size = arrayHeap.size();
      while (2*p + 1 < size)
      {
         int leftChildPos = 2*p + 1;
         int rightChildPos = leftChildPos + 1;
         int minChildPos = leftChildPos;

         // Is there a right child?
         if (rightChildPos < size)
         {
            // Which child is smaller
            if (valueAt(rightChildPos) < valueAt(leftChildPos))
            {
               minChildPos = rightChildPos;
            }
         }
         // If less than children we are done,
         // otherwise swap node with smaller child
         if (valueAt(p) <= valueAt(minChildPos))
            break;
         else
         {
            // Do the swap
            Integer temp = arrayHeap.get(p);
            arrayHeap.set(p, arrayHeap.get(minChildPos));
            arrayHeap.set(minChildPos, temp);
         }
         // Go down to the child position
         p = minChildPos;
      }
   }

   /**
      The add method adds a value to the heap.
      @param x The value to add.
      @return true.
   */

   boolean add(int x)
   {
      // Add x at the end of the array list
      arrayHeap.add(x);

      // Sift up
      siftUp();
      return true;
   }

   /**
      The removeMin method removes an item from the heap.
      @return The minimum element in the heap.
      @exception RuntimeException When priority
      queue is empty.
   */

   public int removeMin()
   {
      if (isEmpty())
      {
         String message = "Priority Queue is empty.";
         throw new RuntimeException(message);
      }
      else
      {
         int val = arrayHeap.get(0);

         // Replace root by last leaf
         int lastPos = arrayHeap.size()-1;
         arrayHeap.set(0, arrayHeap.get(lastPos));

         // Remove the last leaf
         arrayHeap.remove(arrayHeap.size()-1);
         siftDown();
         return val;
      }
   }

   /**
      The isEmpty method checks if the heap is empty.
      @return true if the heap is empty
      and false otherwise.
   */

   public boolean isEmpty()
   {
      return arrayHeap.size() == 0;
   }

   /**
      The valueAt method is a convenience method,
      it makes the code more readable.
      @return value stored at given position in the heap.
   */

   private int valueAt(int pos)
   {
      return arrayHeap.get(pos);
   }
}
```

The `ArrayHeapsort` program, given in Code Listing 21-16 demonstrates the use of the `ArrayHeap` class to sort an array. No output is shown for this program because it is similar to that of other programs in this chapter.

**Code Listing 21-16** (`ArrayHeapsort.java`)

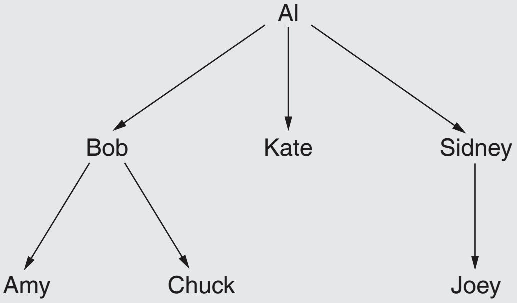
```java
import java.util.*;

/**
   This program demonstrates the use of the
   ArrayHeap class.
*/

public class ArrayHeapsort
{
   public static void main(String [] args)
   {
      // Create and display an array of random integers
      Random randy = new Random();
      int [ ] arr = new int[10];
      System.out.println("Here is the array to be sorted:");
      for (int k = 0; k < arr.length; k++)
      {
         arr[k] = randy.nextInt(100);
         System.out.print(arr[k] + "  ");
      }

      // Create a heap of integers
      // and use it to sort the array
      ArrayHeap heap = new ArrayHeap();
      for (int x : arr) heap.add(x);
      for (int k = 0; k < arr.length; k++)
         arr[k] = heap.removeMin();

      // Print the array
      System.out.println("\nHere is the sorted array:");
      for (int x : arr)
         System.out.print(x + "  ");
   }
}
```

## In the Spotlight:

### Rooted Trees and the `TreeView` Component

A rooted tree is a generalization of a binary tree in which nodes may have more than two children. A familiar example of a rooted tree is the hierarchical file and directory system used on almost all computers. Figure 21-27 shows a simple example of a rooted tree that might be a part of a genealogy tree.

**Figure 21-27** A rooted tree



Many programs need to display rooted trees as part of their user interfaces. JavaFX provides a generic class `TreeItem<T>` to represent a node of a tree. The type parameter `T` represents the type of data stored in the tree item. The following code creates a tree item that will serve as the root of the tree in Figure 21-27:

```
TreeItem<String> root = new TreeItem<>("Al");
```

A tree item can have other tree items as children. The following lines of code create three items and make them children of the one already created:

```
TreeItem<String> bob =  new TreeItem<>("Bob");
TreeItem<String> kate =  new TreeItem<>("Kate");
TreeItem<String> sidney =  new TreeItem<>("Sidney");
root.getChildren().addAll(bob, kate, sidney);
```

To form an in-memory representation of a rooted tree, first create a tree item to serve as the root of the tree, and then add new items as children of those already in the tree. To display the tree as part of the JavaFX scene graph, you must wrap its root item in a `TreeView` component like this:

```
TreeView<String> treeView = new TreeView<>(root);
```

The tree view component can then be added to a `Pane` component that is already part of the scene graph. Alternatively, the tree view can itself be the root of the scene graph. Code Listing 21-17 illustrates the process of building and displaying the tree of Figure 21-28.

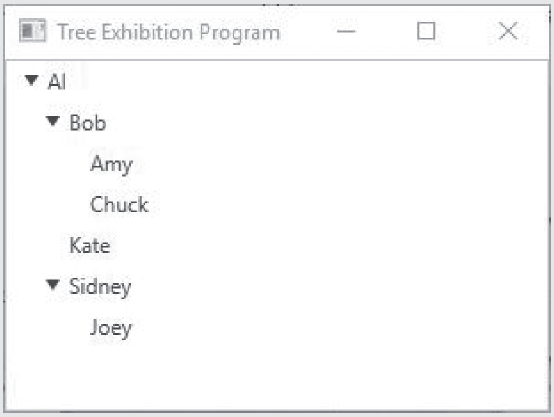
**Code Listing 21-17** `(TreeViewProg.java)`

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.stage.Stage;

/**
```

```
   This program demonstrates use of
   the TreeView component
 */

public class TreeViewProg extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        // Create root node
        TreeItem<String> root = new TreeItem<>("Al");
        root.setExpanded(true);

        // Create three nodes to be children of the root
        TreeItem<String> bob =  new TreeItem<>("Bob");
        TreeItem<String> kate =  new TreeItem<>("Kate");
        TreeItem<String> sidney =  new TreeItem<>("Sidney");
        root.getChildren().addAll(bob, kate, sidney);

        // Create Bob's children
        TreeItem<String> amy =  new TreeItem<>("Amy");
        TreeItem<String> chuck =  new TreeItem<>("Chuck");
        bob.getChildren().addAll(amy, chuck);

        // Create Sidney's child
        TreeItem<String> joey =  new TreeItem<>("Joey");
        sidney.getChildren().add(joey);

        TreeView<String> treeView = new TreeView<>(root);

        Scene scene = new Scene(treeView, 300, 250);
        primaryStage.setTitle("Tree Exhibition Program");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
       @param args the command-line arguments
    */
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

The user interface displayed when this program is run is shown in Figure 21-28.

**Figure 21-28** Screen shot of the `TreeView` example program (Mark Schrier. Copyright Oracle, Inc.)



## 21.5 Common Errors to Avoid

- **Forgetting to check for an empty tree.** Make sure a reference to a node is not `null` before you try to access the value or children of the node.
- **Forgetting to replace a subtree with a modified subtree.** If you have to alter a tree, you will often have to recursively alter one or both subtrees. Make sure that you replace the current subtree with the altered version.
- **Trying to use a loop when recursion is the best way to go.** Most problems on binary trees have simple and elegant recursive solutions. Very often you don't need to use both iteration and recursion on the same problem.

## Review Questions and Exercises

### Multiple Choice and True/False

1. A binary tree is a collection of items in which each item __________.
   a. has no successor
   b. has one successor
   c. has exactly two successors
   d. has at most two successors
2. The three standard techniques for traversing a binary tree are __________.
   a. preorder, inorder, and postorder
   b. prefix, infix, and postfix

c. left-to-right, top-down, and parent-to-child
d. none of the above

3. The set of all descendants of a node in a binary tree form __________.
   a. a path from the node to the end of the tree
   b. a path from the node to the root
   c. a subtree rooted at that node
   d. a set that is best traversed using inorder traversal
4. A binary search tree is structured in such a way that __________.
   a. for each node, the left subtree holds values that are less than the node, and the right subtree holds values that are greater
   b. for each node, the left child holds a value that is less than the node, and the right child holds a value that is greater
   c. the value stored at each node is less than values stored in the children
   d. the heights of the subtrees at any node differ by at most 1
5. An AVL tree is __________.
   a. a kind of priority queue
   b. a kind of heap
   c. a kind of binary search tree
   d. none of the above
6. When a node with two children is removed from a binary search tree, it can be replaced with __________.
   a. the rightmost leaf at the deepest level of the tree
   b. the smallest node in its right subtree
   c. the smallest node in its left subtree
   d. the smallest ancestor of the node
7. In a binary tree that is a heap, __________.
   a. every non-root node must have a value greater than that of its parent
   b. every left child must be less than its parent, and every right child must be greater
   c. the height of the left and right subtrees of each node must differ by at most 1
   d. leaf nodes must be greater or equal to every other node
8. In a complete binary tree, __________.
   a. every level $l$ other than the last must have $2^l$ nodes
   b. the depth $d$ of the tree satisfies $d \leq \log n + 1$, where $n$ is the number of nodes in the tree
   c. every node that has a right child also has a left child
   d. all of the above
9. The Heapsort method __________.
   a. is based on a complete binary search tree structure with the heap order property
   b. makes $O(n \log n)$ comparisons when sorting an array of $n$ entries
   c. is a very good sorting method, but there are other general sorting methods that are much faster
   d. all of the above

10. To add a new value to a heap, you should __________.
    a. add it as a leaf and then sift up
    b. sift the root down to make room for the new node
    c. replace the root with the deepest and rightmost leaf, and then sift down
    d. none of the above
11. **True or False:** A node in a binary tree can have two parents.
12. **True or False:** A node in a binary tree can have no parents.
13. **True or False:** The depth of a binary tree is always the same as the height of its root.
14. **True or False:** The level of a node is the height of the tree minus the height of the node.
15. **True or False:** Removing an item from a priority queue gives you the minimum element.
16. **True or False:** The height of an empty binary tree is not defined.
17. **True or False:** A heap is a complete binary tree that has the heap order property.
18. **True or False:** An ancestor of a node can never be in a subtree of the node.
19. **True or False:** A descendant of a node must belong to at least one subtree of the node.
20. **True or False:** The subtrees of a node in an AVL tree can be equal in height.
21. **True or False:** The subtrees of a node in a complete binary tree must be equal in height.

## Find the Error

Find the error in each of the following code segments.

1.
```
// return number of nodes in a binary tree
int NodeCount(Node tree)
{
   int count = 0;
   if (tree != null)
   {
      count ++;
      NodeCount(tree.left);
      NodeCount(tree.right);
   }
   return count;
}
```

2.
```
// Add a value to a binary search tree
// Return root of resulting search tree
Node add(Node tree, int value)
{
   if (tree == null)
      return new Node(x);
   if (value < tree.value)
      return add(tree.left, value);
   else
      return add(tree.right, value);
}
```

3.
```
// Add 1 to the value in each node of the tree
void incrementAll(Node tree)
{
   incrementAll(tree.left);
   incrementAll(tree.right);
   tree.Value ++;
}
```

### Short Answer

1. Why is it good for binary search trees to be balanced?
2. A binary search tree has a right subtree but no left subtree. What node contains the least element in the tree?
3. What is the height of a binary tree?
4. Explain the concept of *order* as used with priority queues.
5. What is a priority queue?
6. Explain why the depth of a complete binary tree with $n$ nodes is at most $\log n + 1$.
7. How does an AVL tree differ from an ordinary binary search tree?

### Algorithm Workbench

1. Write a method

   `void preorder(Node bTree)`

   that uses a preorder traversal to print all values stored in a binary tree.
2. Write a method

   `void postorder(Node bTree)`

   that uses a postorder traversal to print all values stored in a binary tree.
3. Write a method

   `int treeSize(Node bTree)`

   that computes and returns the number of nodes in a binary tree.
4. Write a method

   `int leafCount(Node btree)`

   that computes and returns the number of leaves in a binary tree.
5. Assume that data is stored in a binary tree, but that unlike the case of a binary search tree, no attempt is made to maintain any sort of order in the data stored. Give an

algorithm for a method `contains` that searches a binary tree for a particular value `x` and returns `true` or `false` according to whether `x` is found in the tree.

6. Define a `Comparator` class `DecOrder` that orders integers in decreasing order, and give the code that shows how to create a `PriorityQueue` object that uses such a `Comparator` object.
7. Define an order on strings where a string `s` is less than `t` if the length of `s` is less than the length of `t`. If `s` and `t` have the same length, then `s` is less than `t` if `s` comes before `t` in alphabetic order. Write a `Comparator` class that corresponds to this order.

## Programming Challenges

### 1. AVL Trees

Add a `remove` method to the AVL tree class of this chapter. Use a graphical user interface that allows a user to work interactively with the AVL tree, adding and removing items as desired.

### 2. Family Tree

Write a program, with a graphical user interface, that allows users to create genealogy trees and work with them. Each person in this genealogy tree will be restricted to having at most two children. The nodes of the tree will store string values, which are interpreted as names of people. The program should support the following commands:

- `root name`: Adds `name` as the root of the tree. This must be the first command entered: it creates a tree with a single node, whose value field holds `name`. The created node becomes the root of the genealogy tree. The command is ignored if it is entered after the tree has been created.
- `left parent child`: Adds a new node `child` as the left child of the node with name `parent`. The command is ignored if there is no node with name `parent` in the tree, or if there is such a node, but it already has a left child.
- `right parent child`: Adds a new node `child` as the right child of the node with name `parent`. The command is ignored if there is no node with name `parent` in the tree, or if there is such a node, but it already has a right child.
- `descendants person`: Displays a list of all descendants of the node with name `person`. The command is ignored if there is no node in the tree with that name.
- `ancestors person`: Displays a list of all ancestors of the node with the name `person`. The command is ignored if there is no node in the tree with that name.

### 3. Balanced Search Tree

Write a program that can convert a sorted array into a balanced binary search tree. For this project, a balanced binary tree is one where the size of the left and right subtrees at each node differs by at most one. Your program should have a graphical user interface. The program allows the user to enter a number *n*, generates an array of *n* random integers, sorts the array, and then converts the sorted array into a balanced binary search tree. The program should display a graphical representation of the binary search tree.

### 4. Prefix Calculator

VideoNote
Prefix Calculator

An arithmetic expression is in *prefix form* when operators are written before their operands. Here are some examples of prefix expressions and the values they evaluate to:

| Expression | Value |
|---|---|
| 12 | 12 |
| + 2 51 | 53 |
| * 5 7 | 35 |
| * + 16 4 + 3 1 | 80 |

An expression (such as 12) that begins with an integer is a prefix expression that evaluates to itself. Otherwise, an expression is a prefix expression if it begins with an operator and is followed by two prefix expressions. In this latter case, the value of the expression is recursively computed from the values of its constituent prefix sub-expressions.

Write a program that allows the user to enter prefix expressions in a text field. The program reads the expression, evaluates it, and displays the value in a suitable GUI component. Assume that the user enters expressions that use only positive integers and the two operators + and *. Your program should internally convert the prefix expression into a binary tree before evaluating it.

### 5. Levels in an AVL Tree

Extend the AVL tree class presented in this chapter by adding a method that takes a value, locates it in the AVL tree, and displays its level. Do this by extending the list of commands supported by the program to include commands of the form *level value*. For example, if the user types in

```
level 12
```

the program displays the level of a node containing 12. The program should display −1 if the user types a value not in the tree. If there is more than one instance of the value in the tree, the program displays the level of any one of them.

### 6. Binary Tree Input and Display

You are given a file that contains data describing a binary tree whose nodes are strings. The file begins with an integer *N*, the number of nodes in the tree, on a line by itself. This first line is followed by *N* additional lines where each line specifies child information for a node in the tree. The line for a node *X* starts with *X* itself followed by a list of the children of *X*. Names of nodes are separated by whitespace. For example, the data set

```
5
Al  Bob  Carol
Bob  Debby  Elaine
Carol
Debby
Elaine
```

represents a binary tree in which Al has two children named Bob and Carol; Bob has two children named Debby and Elaine; and Carol, Debby, and Elaine have no children. Write a program that reads in data from such a file and displays it in a `JTree` component.

### 7. Breadth-First Traversal of Binary trees

Write a program that reads a file that describes a binary tree using the format of Programming Challenge 6. The program then lists all the nodes in the tree by their distance from the root: first the root, then all the children of the root, then all nodes that are children of a child of the root, and so on. For the example tree in Programming Challenge 6, the program might output the nodes in any of the following orders:

```
Al Bob Carol Debby Elaine
Al Carol Bob Elaine Debby
Al Bob Carol Debby Elaine
```

Such ordering are called breadth-first orders. Note that these three are not the only possible breadth-first orders.

# CHAPTER 22 Databases

## TOPICS

22.1 Introduction to Database Management Systems
22.2 Tables, Rows, and Columns
22.3 Introduction to the SQL SELECT Statement
22.4 Inserting Rows
22.5 Updating and Deleting Existing Rows
22.6 Creating and Deleting Tables
22.7 Creating a New Database with JDBC
22.8 Scrollable Result Sets
22.9 Result Set Metadata
22.10 Relational Data
22.11 Advanced Topics
22.12 Common Errors to Avoid

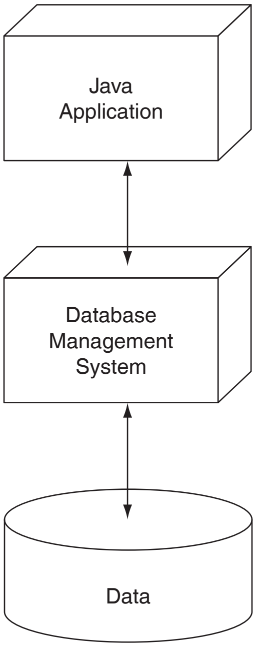
## 22.1 Introduction to Database Management Systems

**CONCEPT: A database management system (DBMS) is software that manages large collections of data.**

If an application needs to store only a small amount of data, text and binary files work well. These types of files, however, are not practical when a large amount of data must be stored and manipulated. Many businesses keep hundreds of thousands, or even millions, of data items in files. When a text or binary file contains this much data, simple operations such as searching, inserting, and deleting become cumbersome and inefficient.

When developing applications that work with an extensive amount of data, most developers prefer to use a database management system. A *database management system* (*DBMS*) is software specifically designed to store, retrieve, and manipulate large amounts of data in an organized and efficient manner. Once the data is stored using the database management system, applications may be written in Java or other languages to communicate with the DBMS. Rather than retrieving or manipulating the data directly, applications can send instructions to the DBMS. The DBMS carries out those instructions and sends the results back to the application. Figure 22-1 illustrates this.

**Figure 22-1** A Java application interacts with a DBMS, which manipulates data



Although Figure 22-1 is simplified, it illustrates the layered nature of an application that works with a database management system. The topmost layer of software, which in this case is written in Java, interacts with the user. It also sends instructions to the next layer of software, the DBMS. The DBMS works directly with the data, and sends the results of operations back to the application.

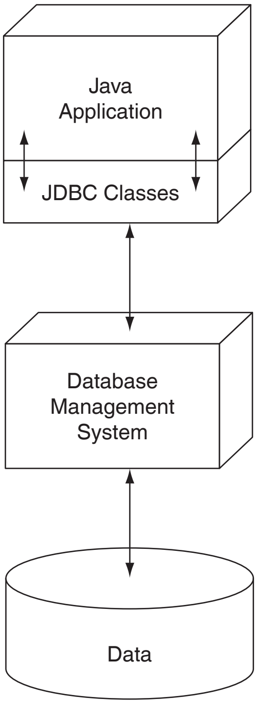
For example, suppose that a company keeps all of its product records in a database. The company has a Java application that allows the user to look up information on any product by entering its product ID number. The Java application instructs the DBMS to retrieve the record for the product with the specified product ID number. The DBMS retrieves the product record and sends the data back to the Java application. The Java application displays the data to the user.

The advantage of this layered approach to software development is that the Java programmer does not need to know about the physical structure of the data. He or she only needs to know how to write code that interacts with the DBMS. The DBMS handles the actual reading, writing, and searching of data.

## JDBC

Figure 22-1 gives a simple illustration of a Java application communicating with a DBMS. The technology that makes this communication possible is known as *JDBC*, which stands for *Java Database Connectivity*. The Java API contains numerous JDBC classes that allow your Java applications to interact with a DBMS. This is illustrated in Figure 22-2.

**Figure 22-2** A Java application uses the JDBC classes to interact with a DBMS



## SQL

*SQL*, which stands for *Structured Query Language*, is a standard language for working with database management systems. It was originally developed by IBM in the 1970s. Since then, SQL has been adopted by most database software vendors as the language of choice for interacting with its DBMS.

SQL consists of several key words. You use the key words to construct statements, which are also known as *queries*. These statements, or queries, are submitted to the DBMS, and are instructions for the DBMS to carry out operations on its data. When a Java application interacts with a DBMS, the Java application must construct SQL statements as strings, and then use an API method to pass those strings to the DBMS. In this chapter, you will learn how to construct simple SQL statements and then pass them to a DBMS using an API method call.

**NOTE:** Although SQL is a language, you don't use it to write applications. It is intended only as a standard means of interacting with a DBMS. You still need a general programming language, such as Java, to write an application for the ordinary user.

## Using a DBMS

In order to use JDBC to work with a database, you will need a DBMS installed on your system, or available to you in a school lab environment. There are many commercial DBMS packages available. Oracle, Microsoft SQL Server, DB2, and MySQL are just a few of the popular ones. In your school's lab, you may already have access to one of these, or perhaps another DBMS.

## Java DB and Apache Derby

If you do not have access to a DBMS in a school lab, you can use either Java DB or Apache Derby, depending on the version of the JDK that you are using. If you are using either Java 7 or Java 8, a DBMS named Java DB is automatically installed on your system when you install the JDK. Java DB is an open source distribution of Apache Derby, a pure Java DBMS that is designed specifically for Java applications.

Beginning with Java 9, however, Java DB is no longer bundled with the JDK. This is because Java DB is simply a rebranded version of Apache Derby. If you are using Java 9, you will need to separately download Apache Derby from the Apache Software Foundation's website and install it.

All of the examples in this chapter were created with Java DB and/or Apache Derby. If you wish to use either Java DB or Apache Derby, see Appendix I, available for download on this book's companion Web site at www.pearson.com/gaddis. Appendix I can help you configure Java DB, or download and install Apache Derby.

## Creating the `CoffeeDB` Database

In this chapter we will use a database named `CoffeeDB` as our example. The `CoffeeDB` database is used in the business operations of The Midnight Roastery, a small coffee roasting company. After you have installed either Java DB or Apache Derby, perform the following steps to create the `CoffeeDB` database:

1. Make sure you have downloaded student source code files from the book's companion Web site.
2. In this chapter's source code files, locate a program named `CreateCoffeeDB.java`.
3. Compile and execute the `CreateCoffeeDB.java` program. If either Java DB or Apache Derby is properly installed, this program will create the `CoffeeDB` database on your system.

> **NOTE:** If you are in a school lab environment using a DBMS other than Java DB or Apache Derby, consult with your instructor on how to modify the program to work with your specific DBMS.

## Connecting to the `CoffeeDB` Database

After installing either Java DB or Apache Derby and creating the `CoffeeDB` database, you should attempt to connect to the database with a Java program. A program can call the static JDBC method `DriverManager.getConnection` to get a connection to a database. There are overloaded versions of this method, but the simplest one has the following general format:

```
DriverManager.getConnection(DatabaseURL);
```

The method returns a reference to a `Connection` object, which we will discuss in a moment. In the general format, *`DatabaseURL`* is a string known as a *database URL*. URL stands for Uniform Resource Locator. A database URL lists the protocol that should be used to access the database, the name of the database, and potentially other items. A simple database URL has the following general format:

```
protocol:subprotocol:databaseName
```

In this very simple general format, three items are listed, separated by colons: *`protocol`*, *`subprotocol`*, and *`databaseName`*. Let's take a closer look at each one.

- *`protocol`* is the database protocol. When using JDBC, the protocol will always be `jdbc`.
- The value for *`subprotocol`* will be dependent upon the particular type of DBMS you are connecting to. If you are using either Java DB or Apache Derby, the subprotocol is `derby`.
- *`databaseName`* is the name of the database you are connecting to.

If we are using either Java DB or Apache Derby the URL for the `CoffeeDB` database is:

```
jdbc:derby:CoffeeDB
```

The `DriverManager.getConnection` method searches for and loads a JDBC driver that is compatible with the database specified by the URL. A *JDBC driver* is a Java class that is designed to communicate with a specific DBMS. Each DBMS usually comes with its own JDBC driver. Typically, when you install a DBMS, you also update your system's `CLASSPATH` variable to include the JDBC driver's location. This will enable the JVM to find the driver class when you call the `DriverManager.getConnection` method.

When the `DriverManager.getConnection` method finds a compatible driver, it returns a `Connection` object. `Connection` is an interface in the `java.sql` package. You will need to use this `Connection` object to perform various tasks with the database, so save the reference in a variable. Here is an example of code that we can use in a Java application to get a connection to the `CoffeeDB` database using either Java DB or Apache Derby:

```
final String DB_URL = "jdbc:derby:CoffeeDB";
Connection conn = DriverManager.getConnection(DB_URL);
```

In the second statement shown here, we call the `DriverManager.getConnection` method, passing the URL for the `CoffeeDB` database. The method returns a reference to a `Connection` object, which we assign to the `conn` variable. If the `DriverManager.getConnection` method fails to load an appropriate driver for the specified database, it will throw an `SQLException`.

Before going any further, compile and execute the `TestConnection.java` program shown in Code Listing 22-1. It demonstrates what we've covered so far. (This program assumes that either Java DB or Apache Derby has been installed, and that the `CoffeeDB` database has been created.)

**Code Listing 22-1** **(`TestConnection.java`)**

```
import java.sql.*;    // Needed for JDBC classes

/**
   This program demonstrates how to connect to
   a Java DB or Apache Derby database using JDBC.
*/

public class TestConnection
{
   public static void main(String[] args)
   {
      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

16        try
17        {
18           // Create a connection to the database.
19           Connection conn = DriverManager.getConnection(DB_URL);
20           System.out.println("Connection created to CoffeeDB.");
21
22           // Close the connection.
23           conn.close();
24           System.out.println("Connection closed.");
25        }
26        catch(Exception ex)
27        {
28           System.out.println("ERROR: " + ex.getMessage());
29        }
30     }
31  }
```

**Program Output**

```
Connection created to CoffeeDB.
Connection closed.
```

Notice that line 1 imports all of the classes in the `java.sql` package. This package contains many of the necessary JDBC classes. Line 14 creates a string constant containing the URL for the `CoffeeDB` database.

JDBC methods throw an `SQLException` if they encounter a problem with a database. For that reason, we use a `try-catch` statement to handle any such exceptions. Let's take a closer look at the statements inside the try block:

- Line 19 does the following:
  - It declares a `Connection` variable named `conn`.
  - It calls the `DriverManager.getConnection` method to get a connection to the `CoffeeDB` database.
  - The `DriverManager.getConnection` method returns a reference to a `Connection` object. The reference is assigned to the `conn` variable.
- Line 20 displays a message indicating that a connection was created.
- Line 23 calls the `Connection` object's `close` method, which simply closes the database connection.
- Line 24 displays a message indicating that the connection is closed.

If a connection cannot be created in line 19, or the connection cannot be closed in line 23, an exception will be thrown. The `catch` clause in line 26 will handle the exception, and line 28 will display the exception object's default error message.

## Connecting to a Password-Protected Database

If the database that you are connecting to requires a user name and a password, you can use the following form of the `DriverManager.getConnection` method:

```
DriverManager.getConnection(DatabaseURL, Username, Password);
```

In this general format, *`Username`* is a string containing a valid username, and *`Password`* is a string containing the password.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

22.1 Why do most businesses use a DBMS to store their data instead of creating their own text files or binary files to hold the data?

22.2 When a Java programmer uses a DBMS to store and manipulate data, why doesn't the programmer need to know specific details about the physical structure of the data?

22.3 What is the technology that makes it possible for a Java application to communicate with a DBMS?

22.4 What is the standard language for working with database management systems?

22.5 What is a database URL?

22.6 Suppose you have either Java DB or Apache Derby database on your system named `InventoryDB`. What database URL would you use in a Java program to get a connection to the database?

22.7 What static JDBC method do you call to get a connection to a database?

## 22.2 Tables, Rows, and Columns

**CONCEPT: Data that is stored in a database is organized into tables, rows, and columns.**

A database management system stores data in a *database*. Your first step in learning to use a DBMS is to learn how data is organized inside a database. The data that is stored in a database is organized into one or more tables. Each *table* holds a collection of related data. The data that is stored in a table is then organized into rows and columns. A row is a complete set of information about a single item. The data that is stored in a row is divided into columns. Each column is an individual piece of information about the item.

The `CoffeeDB` database has a table named `Coffee`, which holds records for all of the different coffees sold by the company. Table 22-1 shows the contents of the table.

**Table 22-1** The `Coffee` database table

| `Description` | `ProdNum` | `Price` |
|---|---|---|
| Bolivian Dark | 14-001 | 8.95 |
| Bolivian Medium | 14-002 | 8.95 |
| Brazilian Dark | 15-001 | 7.95 |
| Brazilian Medium | 15-002 | 7.95 |
| Brazilian Decaf | 15-003 | 8.55 |
| Central American Dark | 16-001 | 9.95 |
| Central American Medium | 16-002 | 9.95 |
| Sumatra Dark | 17-001 | 7.95 |
| Sumatra Decaf | 17-002 | 8.95 |

(*table continues next page*)

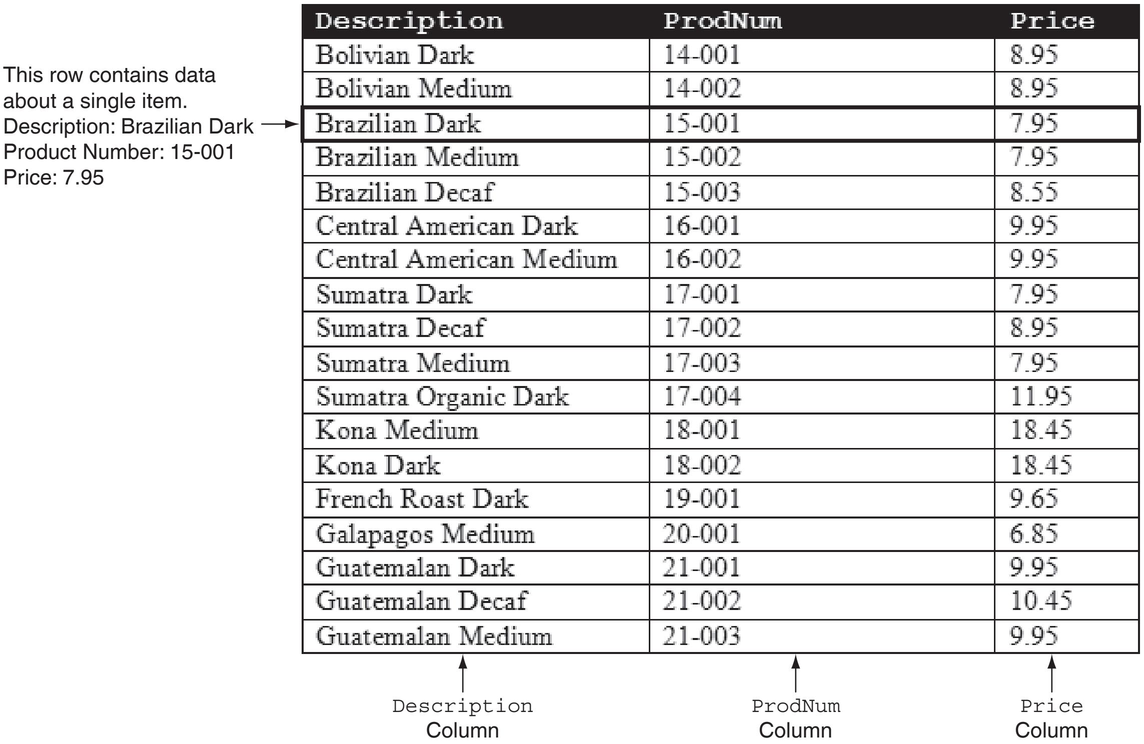
**Table 22-1** The `Coffee` database table (continued)

| `Description` | `ProdNum` | `Price` |
|---|---|---|
| Sumatra Medium | 17-003 | 7.95 |
| Sumatra Organic Dark | 17-004 | 11.95 |
| Kona Medium | 18-001 | 18.45 |
| Kona Dark | 18-002 | 18.45 |
| French Roast Dark | 19-001 | 9.65 |
| Galapagos Medium | 20-001 | 6.85 |
| Guatemalan Dark | 21-001 | 9.95 |
| Guatemalan Decaf | 21-002 | 10.45 |
| Guatemalan Medium | 21-003 | 9.95 |

As you can see, the table has 18 rows. Each row holds data about a type of coffee. The rows are divided into three columns. The first column is named `Description`, and it holds the description of a type of coffee. The second column is named `ProdNum`, and it holds a coffee's product number. The third column is named `Price`, and it holds a coffee's price per pound. As illustrated in Figure 22-3, the third row in the table holds the following data:

Description: Brazilian Dark
Product Number: 15-001
Price: 7.95

**Figure 22-3** The `Coffee` database table



| `Description` | `ProdNum` | `Price` |
|---|---|---|
| Bolivian Dark | 14-001 | 8.95 |
| Bolivian Medium | 14-002 | 8.95 |
| Brazilian Dark | 15-001 | 7.95 |
| Brazilian Medium | 15-002 | 7.95 |
| Brazilian Decaf | 15-003 | 8.55 |
| Central American Dark | 16-001 | 9.95 |
| Central American Medium | 16-002 | 9.95 |
| Sumatra Dark | 17-001 | 7.95 |
| Sumatra Decaf | 17-002 | 8.95 |
| Sumatra Medium | 17-003 | 7.95 |
| Sumatra Organic Dark | 17-004 | 11.95 |
| Kona Medium | 18-001 | 18.45 |
| Kona Dark | 18-002 | 18.45 |
| French Roast Dark | 19-001 | 9.65 |
| Galapagos Medium | 20-001 | 6.85 |
| Guatemalan Dark | 21-001 | 9.95 |
| Guatemalan Decaf | 21-002 | 10.45 |
| Guatemalan Medium | 21-003 | 9.95 |

This row contains data
about a single item.
Description: Brazilian Dark
Product Number: 15-001
Price: 7.95

## Column Data Types

The columns in a database table are assigned a data type. Notice that the `Description` and `ProdNum` columns in the `Coffee` table hold strings, and the `Price` column holds floating-point numbers. The data types of the columns are not Java data types, however. Instead, they are SQL data types. Table 22-2 lists a few of the standard SQL data types, and shows the Java data type that each is generally compatible with.

**Table 22-2** A few of the SQL data types

| SQL Data Type | Description | Corresponding Java Data Type |
|---|---|---|
| `INTEGER` or `INT` | An integer number | `int` |
| `CHARACTER(`*n*`)` or `CHAR(`*n*`)` | A fixed-length string with a length of *n* characters | `String` |
| `VARCHAR(`*n*`)` | A variable-length string with a maximum length of *n* characters | `String` |
| `REAL` | A single-precision floating-point number | `float` |
| `DOUBLE` | A double-precision floating-point number | `double` |
| `DECIMAL(`*t*`, `*d*`)` | A decimal value with *t* total digits and *d* digits appearing after the decimal point | `java.math.BigDecimal` |
| `DATE` | A date | `java.sql.Date` |

There are many other standard data types in SQL. When the `Coffee` table was created, the following data types were used for the columns:

- The data type for the `Description` column is `CHAR(25)`. This means that each value in the `Description` column is a string with a fixed length of 25 characters, compatible with the `String` type in Java.
- The data type for the `ProdNum` column is `CHAR(10)`. This means that each value in the `ProdNum` column is a string with a fixed length of 10 characters, compatible with the `String` type in Java.
- The data type for the `Price` column is `DOUBLE`. This means that each value in the `Price` column is a double-precision floating-point number, compatible with the `double` data type in Java.

## Primary Keys

Most database tables have a *primary key*, which is a column that can be used to identify a specific row in a table. The column that is designated as the primary key holds a unique value for each row. If you try to store duplicate data in the primary key column, an error will occur.

In the `Coffee` table, the `ProdNum` column is the primary key because it holds a unique product number for each type of coffee. Here are some other examples:

- Suppose a table stores employee data, and one of the columns holds employee ID numbers. Because each employee's ID number is unique, this column can be used as the primary key.
- Suppose a table stores data about a cell phone company's inventory of phones, and one of the columns holds cell phone serial numbers. Because each phone's serial number is unique, this column can be used as the primary key.
- Suppose a table stores invoice data, and one of the columns holds invoice numbers. Each invoice has a unique invoice number, so this column can be used as a primary key.

**NOTE:** It is possible for a table's primary key to be the combination of several columns in the table.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

22.8 Describe how the data that is stored in a table is organized.

22.9 What is a primary key?

22.10 What Java data types correspond with the following SQL types?

- `INTEGER`
- `INT`
- `REAL`
- `CHAR`
- `CHARACTER`
- `VARCHAR`
- `DOUBLE`

## 22.3 Introduction to the SQL SELECT Statement

**CONCEPT: The `SELECT` statement is used in SQL to retrieve data from a database.**

The first SQL statement we will discuss is the `SELECT` statement. You use the *`SELECT` statement* to retrieve the rows in a table. As its name implies, the `SELECT` statement allows you to select specific rows. We will start with a very simple form of the statement, as shown here:

```
SELECT Columns FROM Table
```

In the general form, *`Columns`* is one or more column names, and *`Table`* is a table name. Here is an example `SELECT` statement that we might execute on the `CoffeeDB` database:

```
SELECT Description FROM Coffee
```

This statement will retrieve the `Description` column for every row in the `Coffee` table. Figure 22-4 shows an example of the results.

**Figure 22-4** `Description` column

| `Description` |
|---|
| Bolivian Dark |
| Bolivian Medium |
| Brazilian Dark |
| Brazilian Medium |
| Brazilian Decaf |
| Central American Dark |
| Central American Medium |
| Sumatra Dark |
| Sumatra Decaf |
| Sumatra Medium |
| Sumatra Organic Dark |
| Kona Medium |
| Kona Dark |
| French Roast Dark |
| Galapagos Medium |
| Guatemalan Dark |
| Guatemalan Decaf |
| Guatemalan Medium |

Figure 22-4 shows the results of a `SELECT` statement, but what happens to these results? In a Java program, the results of a `SELECT` statement are returned to the program in a `ResultSet` object. A *`ResultSet` object* is simply an object that contains the results of an SQL statement. The process of sending an SQL statement to a DBMS can be summarized in the following steps:

1. Get a connection to the database.
2. Pass a string containing an SQL statement to the DBMS. If the SQL statement has results to send back, a `ResultSet` object will be returned to the program.
3. Process the contents of the `ResultSet` object, if one has been returned to the program.
4. When finished working with the database, close the connection.

You previously saw, in Code Listing 22-1, an example of how to perform step 1 (get a connection to the database) and step 4 (close the connection). Next we look at the details of how an SQL statement is sent to the DBMS and how its results are processed in steps 2 and 3.

## Passing an SQL Statement to the DBMS

Once you have gotten a connection to the database, you are ready to issue SQL statements to the DBMS. First, you must get a `Statement` object from the `Connection` object, using its `createStatement` method. Here is an example:

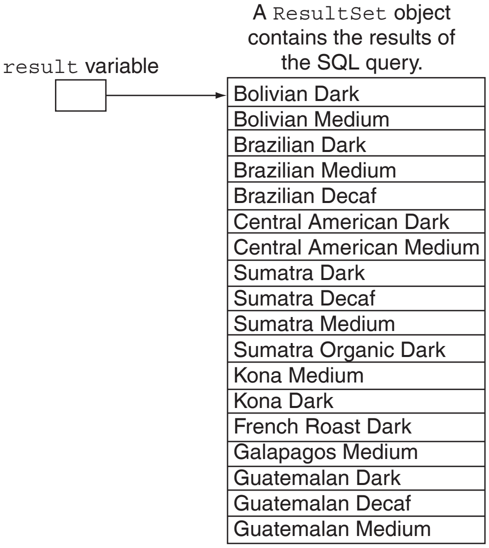
```
Statement stmt = conn.createStatement();
```

After this code executes, the `stmt` variable will reference a `Statement` object. `Statement` is an interface in the `java.sql` package. `Statement` objects have a variety of methods that can be used to execute SQL queries. To execute a `SELECT` query, you use the `executeQuery` method. The method returns a `ResultSet` object. Here is an example:

```
String sqlStatement = "SELECT Description FROM Coffee";
ResultSet result = stmt.executeQuery(sqlStatement);
```

The first statement creates a string containing an SQL query. The second statement passes this string as an argument to the `executeQuery` method. The method returns a reference to a `ResultSet` object containing the results of the query. The reference is assigned to the `result` variable. Figure 22-5 illustrates how the `result` variable references the `ResultSet` object.

**Figure 22-5** A `ResultSet` object contains the results of an SQL query


A ResultSet object contains the results of the SQL query.
result variable
Bolivian Dark
Bolivian Medium
Brazilian Dark
Brazilian Medium
Brazilian Decaf
Central American Dark
Central American Medium
Sumatra Dark
Sumatra Decaf
Sumatra Medium
Sumatra Organic Dark
Kona Medium
Kona Dark
French Roast Dark
Galapagos Medium
Guatemalan Dark
Guatemalan Decaf
Guatemalan Medium

A `ResultSet` object contains a set of rows and columns. The `ResultSet` object in Figure 22-5 has 18 rows and one column. The rows in a `ResultSet` are numbered, with the first row being row 1, the second row being row 2, and so forth. The columns also are numbered, with the first column being column 1, the second column being column 2, and so forth. Figure 22-6 shows the same `ResultSet` with the row and column numbers labeled.

**Figure 22-6** `ResultSet` rows and columns

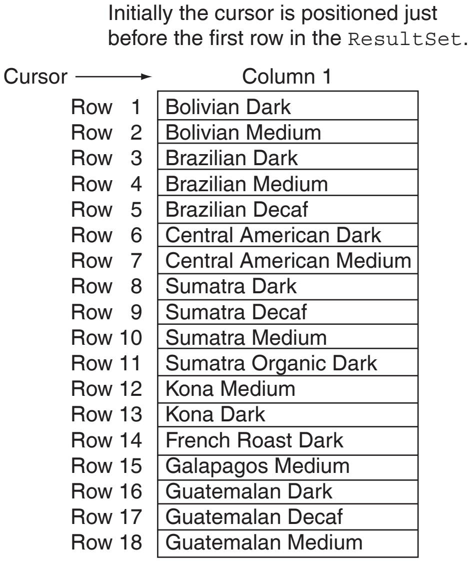
|  | Column 1 |
|---|---|
| Row 1 | Bolivian Dark |
| Row 2 | Bolivian Medium |
| Row 3 | Brazilian Dark |
| Row 4 | Brazilian Medium |
| Row 5 | Brazilian Decaf |
| Row 6 | Central American Dark |
| Row 7 | Central American Medium |
| Row 8 | Sumatra Dark |
| Row 9 | Sumatra Decaf |
| Row 10 | Sumatra Medium |
| Row 11 | Sumatra Organic Dark |
| Row 12 | Kona Medium |
| Row 13 | Kona Dark |
| Row 14 | French Roast Dark |
| Row 15 | Galapagos Medium |
| Row 16 | Guatemalan Dark |
| Row 17 | Guatemalan Decaf |
| Row 18 | Guatemalan Medium |

### Getting a Row from the `ResultSet` Object

A `ResultSet` object has an internal *cursor* that points to a specific row in the `ResultSet`. The row that the cursor points to is considered the *current row*. The cursor can be moved from row to row, and this provides you with a way to examine all of the rows in the `ResultSet`.

At first, the cursor is not pointing to a row, but is positioned just before the first row. This is illustrated in Figure 22-7.

**Figure 22-7** The cursor is initially positioned before the first row



Initially the cursor is positioned just before the first row in the `ResultSet`.

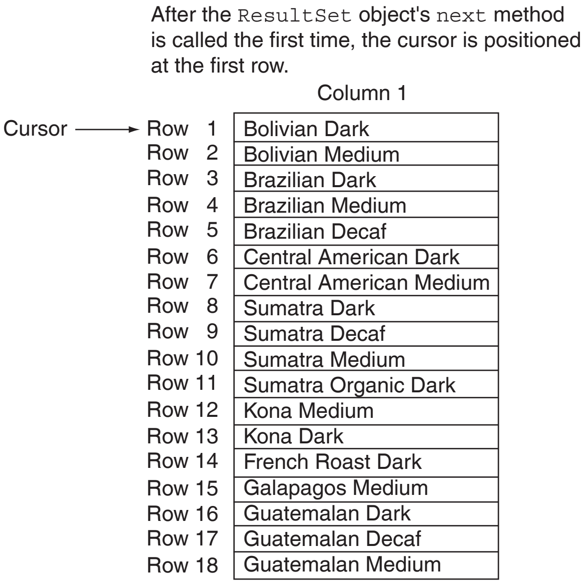
|  | Column 1 |
|---|---|
| Row 1 | Bolivian Dark |
| Row 2 | Bolivian Medium |
| Row 3 | Brazilian Dark |
| Row 4 | Brazilian Medium |
| Row 5 | Brazilian Decaf |
| Row 6 | Central American Dark |
| Row 7 | Central American Medium |
| Row 8 | Sumatra Dark |
| Row 9 | Sumatra Decaf |
| Row 10 | Sumatra Medium |
| Row 11 | Sumatra Organic Dark |
| Row 12 | Kona Medium |
| Row 13 | Kona Dark |
| Row 14 | French Roast Dark |
| Row 15 | Galapagos Medium |
| Row 16 | Guatemalan Dark |
| Row 17 | Guatemalan Decaf |
| Row 18 | Guatemalan Medium |

To move the cursor to the first row in a newly created `ResultSet`, you call the object's `next` method. Here is an example:

```
result.next();
```

Assuming that `result` references a newly created `ResultSet` object, this statement moves the cursor to the first row in the `ResultSet`. Figure 22-8 shows how the cursor has moved to the first row in the `ResultSet` after the `next` method is called the first time.

**Figure 22-8** The `next` method moves the cursor forward



After the `ResultSet` object's `next` method is called the first time, the cursor is positioned at the first row.

| | Column 1 |
|---|---|
| Row 1 | Bolivian Dark |
| Row 2 | Bolivian Medium |
| Row 3 | Brazilian Dark |
| Row 4 | Brazilian Medium |
| Row 5 | Brazilian Decaf |
| Row 6 | Central American Dark |
| Row 7 | Central American Medium |
| Row 8 | Sumatra Dark |
| Row 9 | Sumatra Decaf |
| Row 10 | Sumatra Medium |
| Row 11 | Sumatra Organic Dark |
| Row 12 | Kona Medium |
| Row 13 | Kona Dark |
| Row 14 | French Roast Dark |
| Row 15 | Galapagos Medium |
| Row 16 | Guatemalan Dark |
| Row 17 | Guatemalan Decaf |
| Row 18 | Guatemalan Medium |

Each time you call the `next` method, it moves the cursor to the next row in the `ResultSet`.

The `next` method returns a `boolean` value. It returns `true` if the cursor successfully moved to the next row. If there are no more rows, it returns `false`. The following code shows how you can move the cursor through all of the rows of a newly created `ResultSet`.

```
while (result.next())
{
   // Process the current row.
}
```

There are other `ResultSet` methods for navigating the rows in a `ResultSet` object. We will look at some of them later in this chapter.

### Getting Columns in a `ResultSet` Row

You use one of the `ResultSet` object's "get" methods to retrieve the contents of a specific column in the current row. When you call one of these methods, you can pass either the column number or the column name as an argument. There are numerous "get" methods defined in the `ResultSet` interface. Table 22-3 lists a few of them.

**Table 22-3** A few of the `ResultSet` methods

| `ResultSet` Method | Description |
|---|---|
| `double getDouble(int colNumber)`<br>`double getDouble(String colName)` | Returns the `double` that is stored in the column specified by *`colNumber`* or *`colName`*. The column must hold data that is compatible with the `double` data type in Java. If an error occurs, the method throws an `SQLException`. |
| `int getInt(int colNumber)`<br>`int getInt(String colName)` | Returns the `int` that is stored in the column specified by *`colNumber`* or *`colName`*. The column must hold data that is compatible with the `int` data type in Java. If an error occurs, the method throws an `SQLException`. |
| `String getString(int colNumber)`<br>`String getString(String colName)` | Returns the string that is stored in the column specified by *`colNumber`* or *`colName`*. The column must hold data that is compatible with the `String` type in Java. If an error occurs, the method throws an `SQLException`. |

Recall that columns have an SQL data type. The SQL data types are not the same as the Java data types, but are compatible with them. To retrieve the contents of a column, you call the method that is designed to work with that column's data type. For example, if the column contains a string, you would use the `getString` method to retrieve its value. If the column contains an integer, you would use the `getInt` method. Likewise, if the column contains a `double`, you would call the `getDouble` method.

When you call one of the "get" methods, you must tell it which column in the current row you want to retrieve. These methods accept either an integer argument, which is a column number, or a `String` holding the column name.

The `ResultSet` that we have been looking at in our example has only one column: the `Description` column. The `Description` column's data type is `CHAR(25)`, which means it is a fixed-length string of 25 characters. This is compatible with the `String` type in Java. To display the contents of the `Description` column in the current row, we could use the following statement:

```
System.out.println(result.getString("Description"));
```

The `Description` column holds values that are compatible with the `String` type, so we use the `getString` method to retrieve its contents. We could also use the column number to retrieve the column contents. Here is an example:

```
System.out.println(result.getString(1));
```

**NOTE:** Column names in a database table are not case sensitive. The column names `DESCRIPTION`, `description`, and `Description` are all the same.

Let's look at a complete program that demonstrates what we have covered so far. Code Listing 22-2 displays the `Description` column from all of the rows in the `CoffeeDB` database.

**Code Listing 22-2** (ShowCoffeeDescriptions.java)

```
import java.sql.*; // Needed for JDBC classes

/**
   This program demonstrates how to issue an SQL
   SELECT statement to a database using JDBC.
*/

public class ShowCoffeeDescriptions
{
   public static void main(String[] args)
   {
      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Create a string with a SELECT statement.
         String sqlStatement = "SELECT Description FROM Coffee";

         // Send the statement to the DBMS.
         ResultSet result = stmt.executeQuery(sqlStatement);

         // Display a header for the listing.
         System.out.println("Coffees Found in the Database");
         System.out.println("-----------------------------");

         // Display the contents of the result set.
         // The result set will have three columns.
         while (result.next())
         {
            System.out.println(result.getString("Description"));
         }

         // Close the connection.
         conn.close();
      }
      catch(Exception ex)
45          {
46             System.out.println("ERROR: " + ex.getMessage());
47          }
48       }
49    }
```

**Program Output**

```
Coffees Found in the Database
-----------------------------
Bolivian Dark
Bolivian Medium
Brazilian Dark
Brazilian Medium
Brazilian Decaf
Central American Dark
Central American Medium
Sumatra Dark
Sumatra Decaf
Sumatra Medium
Sumatra Organic Dark
Kona Medium
Kona Dark
French Roast Dark
Galapagos Medium
Guatemalan Dark
Guatemalan Decaf
Guatemalan Medium
```

Let's take a closer look at the code. Line 14 declares a string constant, initialized with the URL for the `CoffeeDB` database. The statements that access the database are written inside the try block that appears in lines 18 through 42. Line 19 gets a connection to the database. After line 19 executes, the `conn` variable will reference a `Connection` object that can be used to access the database.

At this point in the program, we have a connection to the `CoffeeDB` database, but we are not ready to send a `SELECT` statement to the database. In order to send a `SELECT` statement to the database, we must have a `Statement` object. Line 22 calls the `Connection` object's `createStatement` method, which returns a reference to a `Statement` object. The reference is assigned to the `stmt` variable.

Line 25 declares a `String` variable named `sqlStatement`, initialized with the string `"SELECT Description FROM Coffee"`. This is the SQL statement that we want to submit to the database. Line 28 passes this string as an argument to the `Statement` object's `executeQuery` method, which executes the statement. The method returns a reference to a `ResultSet` object, which is assigned to the result variable. The `ResultSet` object contains the results of the `SELECT` statement.

The `while` loop that appears in lines 36 through 39 displays the contents of the `ResultSet` object. It works like this:

- The `while` statement in line 36 calls the `ResultSet` object's `next` method to advance the internal cursor. If the cursor is successfully advanced, the method returns `true` and the loop iterates. If the cursor is at the end of the `ResultSet` object's rows, the method returns `false` and the loop terminates.
- Each time the loop iterates, the `ResultSet` object's internal cursor will be positioned at a specific row. The statement in line 38 gets the value of the `Description` column and displays it.

Line 42 closes the connection to the database.

### More about the SELECT Statement

You can specify more than one column in a `SELECT` statement by separating the column names with commas. Here is an example:

```
SELECT Description, Price FROM Coffee
```

This statement will retrieve the `Description` column and the `Price` column for every row in the `Coffee` table. The program shown in Code Listing 22-3 demonstrates.

**Code Listing 22-3** (`ShowDescriptionsAndPrices.java`)

```
 1  import java.sql.*;  // Needed for JDBC classes
 2
 3  /**
 4     This program displays the coffee descriptions
 5     and their prices.
 6  */
 7
 8  public class ShowDescriptionsAndPrices
 9  {
10     public static void main(String[] args)
11     {
12        // Create a named constant for the URL.
13        // NOTE: This value is specific for either Java DB or Apache Derby.
14        final String DB_URL = "jdbc:derby:CoffeeDB";
15
16        try
17        {
18           // Create a connection to the database.
19           Connection conn = DriverManager.getConnection(DB_URL);
20
21           // Create a Statement object.
22           Statement stmt = conn.createStatement();
23
24           // Create a string with a SELECT statement.
25           String sqlStatement =
               "SELECT Description, Price FROM Coffee";

            // Send the statement to the DBMS.
            ResultSet result = stmt.executeQuery(sqlStatement);

            // Display the contents of the result set.
            // The result set will have three columns.
            while (result.next())
            {
               System.out.printf("%25s %.2f\n",
                                 result.getString("Description"),
                                 result.getDouble("Price"));
            }

            // Close the connection.
            conn.close();
        }
        catch(Exception ex)
        {
           System.out.println("ERROR: " + ex.getMessage());
        }
     }
  }
```

**Program Output**

```
Bolivian Dark                      8.95
Bolivian Medium                    8.95
Brazilian Dark                     7.95
Brazilian Medium                   7.95
Brazilian Decaf                    8.55
Central American Dark              9.95
Central American Medium            9.95
Sumatra Dark                       7.95
Sumatra Decaf                      8.95
Sumatra Medium                     7.95
Sumatra Organic Dark               11.95
Kona Medium                        18.45
Kona Dark                          18.45
French Roast Dark                  9.65
Galapagos Medium                   6.85
Guatemalan Dark                    9.95
Guatemalan Decaf                   10.45
Guatemalan Medium                  9.95
```

The program in Code Listing 22-3 is very similar to that in Code Listing 22-2. The differences between the two programs are summarized here:

- Lines 25 and 26 initialize the `sqlStatement` variable with the string `"SELECT Description, Price FROM Coffee"`. This is a `SELECT` statement that will retrieve the `Description` and `Price` columns from the database table.
- Inside the `while` loop, in lines 35 through 37, we call `result.getString` to get the current row's `Description` column, and we call `result.getDouble` to get the current row's `Price` column. These items are displayed with the `System.out.printf` method.

If you wish to retrieve every column, you can use the * character instead of listing column names. Here is an example:

```
SELECT * FROM Coffee
```

This statement will retrieve every column for every row in the `Coffee` table. The program shown in Code Listing 22-4 demonstrates.

**Code Listing 22-4** `(ShowCoffeeData.java)`

```
import java.sql.*;  // Needed for JDBC classes

/**
   This program displays all of the columns in the
   Coffee table of the CoffeeDB database.
*/

public class ShowCoffeeData
{
   public static void main(String[] args)
   {
      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Create a string with a SELECT statement.
         String sqlStatement = "SELECT * FROM Coffee";

         // Send the statement to the DBMS.
         ResultSet result = stmt.executeQuery(sqlStatement);

         // Display the contents of the result set.
         // The result set will have three columns.
```

```
          while (result.next())
          {
             System.out.printf("%25s %10s %5.2f\n",
                               result.getString("Description"),
                               result.getString("ProdNum"),
                               result.getDouble("Price"));
          }

          // Close the connection.
          conn.close();
      }
      catch(Exception ex)
      {
         System.out.println("ERROR: " + ex.getMessage());
      }
   }
}
```

**Program Output**

```
Bolivian Dark                       14-001          8.95
Bolivian Medium                     14-002          8.95
Brazilian Dark                      15-001          7.95
Brazilian Medium                    15-002          7.95
Brazilian Decaf                     15-003          8.55
Central American Dark               16-001          9.95
Central American Medium             16-002          9.95
Sumatra Dark                        17-001          7.95
Sumatra Decaf                       17-002          8.95
Sumatra Medium                      17-003          7.95
Sumatra Organic Dark                17-004          11.95
Kona Medium                         18-001          18.45
Kona Dark                           18-002          18.45
French Roast Dark                   19-001          9.65
Galapagos Medium                    20-001          6.85
Guatemalan Dark                     21-001          9.95
Guatemalan Decaf                    21-002          10.45
Guatemalan Medium                   21-003          9.95
```

SQL statements are free-form, which means that tabs, newlines, and spaces between the key words are ignored. For example, the statement

```
SELECT * FROM Coffee
```

works the same as:

```
SELECT
   *
FROM
   Coffee
```

In addition, SQL key words and table names are case insensitive. The previous statement could be written as:

```
select * from coffee
```

## Specifying Search Criteria with the `WHERE` Clause

Occasionally you might want to retrieve every row in a table. For example, if you wanted a list of all the coffees in the `Coffee` table, the previous `SELECT` statement would give it to you. Normally, however, you want to narrow the list down to only a few selected rows in the table. That's where the `WHERE` clause comes in. The `WHERE` clause can be used with the `SELECT` statement to specify search criteria. When you use the `WHERE` clause, only the rows that meet the search criteria will be returned in the result set. The general format of a `SELECT` statement with a `WHERE` clause is:

```
SELECT Columns FROM Table WHERE Criteria
```

In the general format, *`Criteria`* is a conditional expression. Here is an example of a `SELECT` statement that uses the `WHERE` clause:

```
SELECT * FROM Coffee WHERE Price > 12.00
```

The first part of the statement, `SELECT * FROM Coffee`, specifies that we want to see every column. The `WHERE` clause specifies that we want only the rows in which the content of the `Price` column is greater than 12.00. Figure 22-9 shows the results of this statement. Notice that only two coffees meet this search criterion.

**Figure 22-9** Rows where `Price` is greater than 12.00

| Description | ProdNum | Price |
|---|---|---|
| Kona Medium | 18-001 | 18.45 |
| Kona Dark | 18-002 | 18.45 |

Standard SQL supports the relational operators listed in Table 22-4 for writing conditional expressions in a `WHERE` clause.

**Table 22-4** SQL relational operators

| Operator | Meaning |
|---|---|
| > | Greater-than |
| < | Less-than |
| >= | Greater-than or equal-to |
| <= | Less-than or equal-to |
| = | Equal-to |
| <> | Not equal-to |

Notice that the equal-to and not equal-to operators in SQL are different from those in Java. The equal-to operator is one equal sign, not two equal signs. The not equal-to operator is <>.

Let's look at a few more examples of the `SELECT` statement. The following statement could be used to retrieve the product numbers and prices of all the coffees that are priced at 7.95:

```
SELECT ProdNum, Price FROM Coffee WHERE Price = 7.95
```

The results of this statement are shown in Figure 22-10.

**Figure 22-10** Results of SQL statement

| ProdNum | Price |
|---|---|
| 16-001 | 7.95 |
| 16-002 | 7.95 |
| 17-001 | 7.95 |
| 17-003 | 7.95 |

The following `SELECT` statement retrieves all of the columns for the row where the description is "French Roast Dark". The results returned from this statement are shown in Figure 22-11.

```
SELECT * FROM Coffee WHERE Description = 'French Roast Dark'
```

**Figure 22-11** Results of SQL statement

| Description | ProdNum | Price |
|---|---|---|
| French Roast Dark | 19-001 | 9.65 |

If you look carefully at the previous statement you will notice another difference between SQL syntax and Java syntax. In SQL, string literals are enclosed in single quotes, not double quotes.

**TIP:** If you need to include a single quote as part of a string, simply write two single quotes in its place. For example, suppose you wanted to search the `Coffee` table for a coffee named Joe's Special Blend. You could use the following statement:

```
SELECT * FROM Coffee WHERE Description = 'Joe''s Special Blend'
```

Let's look at an example program that uses a `WHERE` clause in a `SELECT` statement. The program in Code Listing 22-5 lets the user enter a minimum price, and then search the `Coffee` table for rows where the `Price` column is greater than or equal to the specified price.

**Code Listing 22-5** (CoffeeMinPrice.java)

```
import java.util.Scanner;
import java.sql.*;

/**
   This program lets the user search for coffees
   priced at a minimum value.
*/

public class CoffeeMinPrice
{
   public static void main(String[] args)
   {
      double minPrice;     // To hold the minimum price
      int coffeeCount = 0; // To count coffees that qualify

      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the minimum price from the user.
      System.out.print("Enter the minimum price: ");
      minPrice = keyboard.nextDouble();

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Create a string containing a SELECT statement.
         // Note that we are incorporating the user's input
         // into the string.
         String sqlStatement =
            "SELECT * FROM Coffee WHERE Price >= " +
            Double.toString(minPrice);

         // Send the statement to the DBMS.
         ResultSet result = stmt.executeQuery(sqlStatement);

         // Display the contents of the result set.
         // The result set will have three columns.
47            while (result.next())
48            {
49               // Display a row from the result set.
50               System.out.printf("%25s %10s %5.2f\n",
51                                 result.getString("Description"),
52                                 result.getString("ProdNum"),
53                                 result.getDouble("Price"));
54
55               // Increment the counter.
56               coffeeCount++;
57            }
58
59            // Display the number of qualifying coffees.
60            System.out.println(coffeeCount + " coffees found.");
61
62            // Close the connection.
63            conn.close();
64         }
65         catch(Exception ex)
66         {
67            System.out.println("ERROR: " + ex.getMessage());
68         }
69      }
70   }
```

**Program Output**

```
Enter the minimum price: 12.00 [Enter]
Kona Medium              18-001       18.45
Kona Dark                18-002       18.45
2 coffees found.
```

**Program Output**

```
Enter the minimum price: 10.00 [Enter]
Sumatra Organic Dark     17-004       11.95
Kona Medium              18-001       18.45
Kona Dark                18-002       18.45
Guatemalan Decaf         21-002       10.45
4 coffees found.
```

**Program Output**

```
Enter the minimum price: 20.00 [Enter]
0 coffees found.
```

There are a few things in Code Listing 22-5 that deserve some explanation. In lines 24 and 25 the program prompts the user to enter a minimum price, and the user's input is assigned to the `double` variable `minPrice`. Then, notice in lines 38 through 40 that the `minPrice` variable is converted to a string, and concatenated onto the string containing the `SELECT` statement. When the program runs, if the user enters 10.00, the `SELECT` statement that is created in lines 38 through 40 will be:

```
SELECT * FROM Coffee WHERE Price >= 10.00
```

Or, if the user enters 12.00, the `SELECT` statement that is created in lines 38 through 40 will be:

```
SELECT * FROM Coffee WHERE Price >= 12.00
```

Programs commonly need to use techniques such as this to create SQL statements that incorporate user input.

### String Comparisons in a `SELECT` Statement

String comparisons in SQL are case sensitive. If you ran the following statement against the `CoffeeDB` database, you would not get any results:

```
SELECT * FROM Coffee WHERE Description = 'french roast dark'
```

However, you can use the `UPPER()` function to convert a string to uppercase. Here is an example:

```
SELECT * FROM Coffee WHERE UPPER(Description) = 'FRENCH ROAST DARK'
```

This statement will return the same results as shown in Figure 22-11. SQL also provides a `LOWER()` function, which converts its argument to lowercase.

### Using the `LIKE` Operator

Sometimes searching for an exact string will not yield the results you want. For example, suppose we want a list of all the decaf coffees in the `Coffee` table. The following statement will not work. Can you see why?

```
SELECT * FROM Coffee WHERE Description = 'Decaf'
```

This statement will search for rows where the `Description` field is equal to the string "Decaf". Unfortunately, it will find none. If you glance back at Table 22-1, you will see that none of the rows in the `Coffee` table have a `Description` column that is equal to "Decaf". You will also see, however, that the word "Decaf" does appear in the `Description` column of some of the rows. For example, in one row you will find "Brazilian Decaf". In another row you will find "Sumatra Decaf". In yet another row you will find "Guatemalan Decaf". In addition to the word "Decaf", each of these strings contains other characters.

In order to find all of the decaf coffees, we need to search for rows where "Decaf" appears as a substring in the `Description` column. You can perform such a search using the `LIKE` operator. Here is an example of how to use it.

```
SELECT * FROM Coffee WHERE Description LIKE '%Decaf%'
```

The `LIKE` operator is followed by a string that contains a character pattern. In this example, the character pattern is `'%Decaf%'`. The `%` symbol is used as a wildcard character. It represents any sequence of zero or more characters. The pattern `'%Decaf%'` specifies that the string "Decaf" must appear with any sequence of characters before or after it. The results of this statement are shown in Figure 22-12.

**Figure 22-12** Results of SQL statement

| Description | ProdNum | Price |
|---|---|---|
| Brazilian Decaf | 15-003 | 8.55 |
| Sumatra Decaf | 17-002 | 8.95 |
| Guatemalan Decaf | 21-002 | 10.45 |

Likewise, the following statement will result in all the rows where the `Description` column starts with the word "Sumatra".

```
SELECT * FROM Coffee WHERE Description LIKE 'Sumatra%'
```

The underscore character ( _ ) is also used as a wildcard. Unlike the `%` character, the underscore represents a single character. For example, look at the following statement.

```
SELECT * FROM Coffee WHERE ProdNum LIKE '2_-00_'
```

This statement will result in all the rows where the `ProdNum` column begins with "2", followed by any single character, followed by "-00", followed by any single character. The results of this statement are shown in Figure 22-13.

**Figure 22-13** Results of SQL statement

| Description | ProdNum | Price |
|---|---|---|
| Galapagos Medium | 20-001 | 6.85 |
| Guatemalan Dark | 21-001 | 9.95 |
| Guatemalan Decaf | 21-002 | 10.45 |
| Guatemalan Medium | 21-003 | 9.95 |

You can use the `NOT` operator to disqualify a character pattern in a search criterion. For example, suppose that you want a list of all the coffees that are not decaf. The following statement will yield just those results.

```
SELECT * FROM Coffee WHERE Description NOT LIKE '%Decaf%'
```

### Using AND and OR

You can use the `AND` and `OR` logical operators to specify multiple search criteria in a `WHERE` clause. For example, look at the following statement:

```
SELECT * FROM Coffee WHERE Price > 10.00 AND Price < 14.00
```

The `AND` operator requires that both of the search criteria be true in order for a row to be qualified as a match. The only rows that will be returned from this statement are those where the `Price` column contains a value that is greater than 10.00 and less than 14.00. Figure 22-14 shows the results of the statement.

**Figure 22-14** Results of SQL statement

| Description | ProdNum | Price |
|---|---|---|
| Sumatra Organic Dark | 17-004 | 11.95 |
| Guatemalan Decaf | 21-002 | 10.45 |

Here's an example that uses the `OR` operator:

```
SELECT * FROM Coffee
WHERE Description LIKE '%Dark%' OR ProdNum LIKE '16%'
```

The `OR` operator requires that either of the search criteria be true in order for a row to be qualified as a match. This statement searches for rows where the `Description` column contains the string "Dark" at any position, or where the `ProdNum` column starts with "16". Figure 22-15 shows the results of the query.

**Figure 22-15** Results of SQL statement

| Description | ProdNum | Price |
|---|---|---|
| Bolivian Dark | 14-001 | 8.95 |
| Brazilian Dark | 15-001 | 7.95 |
| Central American Dark | 16-001 | 9.95 |
| Central American Medium | 16-002 | 9.95 |
| Sumatra Dark | 17-001 | 7.95 |
| Sumatra Organic Dark | 17-004 | 11.95 |
| Kona Dark | 18-002 | 18.45 |
| French Roast Dark | 19-001 | 9.65 |
| Guatemalan Dark | 21-001 | 9.95 |

## Sorting the Results of a SELECT Query

If you wish to sort the results of a `SELECT` query, you can use the `ORDER BY` clause. Here is an example:

```
SELECT * FROM Coffee ORDER BY Price
```

This statement will produce a list of all the rows in the `Coffee` table, ordered by the `Price` column. The list will be sorted in ascending order on the `Price` column, meaning that the lowest-priced coffees will appear first.

Here's a `SELECT` statement that uses both a `WHERE` clause and an `ORDER BY` clause:

```
SELECT * FROM Coffee
  WHERE Price > 9.95
  ORDER BY Price
```

This statement will produce a list of all the rows in the `Coffee` table where the `Price` column contains a value greater than 9.95, listed in ascending order by price.

If you want the list sorted in descending order (from highest to lowest), use the DESC operator, as shown here:

```
SELECT * FROM Coffee
  WHERE Price > 9.95
  ORDER BY Price DESC
```

## Mathematical Functions

SQL provides several functions for performing calculations. For example, the AVG function calculates the average value in a particular column. Here is an example SELECT statement using the AVG function:

```
SELECT AVG(Price) FROM Coffee
```

This statement produces a single value: the average of all the values in the Price column. Because we did not use a WHERE clause, it uses all of the rows in the Coffee table in the calculation. Here is an example that calculates the average price of all the coffees having a product number that begins with "20":

```
SELECT AVG(Price) FROM Coffee WHERE ProdNum LIKE '20%'
```

Another of the mathematical functions is SUM, which calculates the sum of a column's values. The following statement, which is probably not very useful, calculates the sum of the values in the Price column:

```
SELECT SUM(Price) FROM Coffee
```

The MIN and MAX functions determine the minimum and maximum values found in a column. The following statement will tell us the minimum value in the Price column:

```
SELECT MIN(Price) FROM Coffee
```

The following statement will tell us the maximum value in the Price column:

```
SELECT MAX(Price) FROM Coffee
```

The COUNT function can be used to determine the number of rows in a table, as demonstrated by the following statement:

```
SELECT COUNT(*) FROM Coffee
```

The * simply indicates that you want to count entire rows. Here is another example, which tells us the number of coffees having a price greater than 9.95:

```
SELECT COUNT(*) FROM Coffee WHERE Price > 9.95
```

Queries that use math functions, such as the examples shown here, return only one value. So, when you submit such a statement to a DBMS using JDBC, the ResultSet object that is returned to the program will contain one row with one column. The program shown in Code Listing 22-6 shows an example of how you can use the MIN, MAX, and AVG functions to find the lowest, highest, and average prices in the Coffee table.

**Code Listing 22-6** (`CoffeeMath.java`)

```
import java.sql.*;

/**
   This program demonstrates some of the SQL math functions.
*/

public class CoffeeMath
{
   public static void main(String[] args)
   {
      // Variables to hold the lowest, highest, and
      // average prices of coffee.
      double lowest = 0.0,
             highest = 0.0,
             average = 0.0;

      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Create SELECT statements to get the lowest, highest,
         // and average prices from the Coffee table.
         String minStatement = "SELECT MIN(Price) FROM Coffee";
         String maxStatement = "SELECT MAX(Price) FROM Coffee";
         String avgStatement = "SELECT AVG(Price) FROM Coffee";

         // Get the lowest price.
         ResultSet minResult = stmt.executeQuery(minStatement);
         if (minResult.next())
            lowest = minResult.getDouble(1);

         // Get the highest price.
         ResultSet maxResult = stmt.executeQuery(maxStatement);
         if (maxResult.next())
            highest = maxResult.getDouble(1);

         // Get the average price.
         ResultSet avgResult = stmt.executeQuery(avgStatement);
47          if (avgResult.next())
48             average = avgResult.getDouble(1);
49
50          // Display the results.
51          System.out.printf("Lowest price: $%.2f\n", lowest);
52          System.out.printf("Highest price: $%.2f\n", highest);
53          System.out.printf("Average price: $%.2f\n", average);
54
55          // Close the connection.
56          conn.close();
57       }
58       catch(Exception ex)
59       {
60          System.out.println("ERROR: " + ex.getMessage());
61       }
62    }
63 }
```

**Program Output**

```
Lowest price: $6.85
Highest price: $18.45
Average price: $10.16
```

Lines 31 through 33 declare three strings: `minStatement`, `maxStatement`, and `avgStatement`. Each of these strings contains a `SELECT` statement that uses a math function.

The code in lines 36 through 38 gets the lowest price in the table. Here is a summary of how the code works:

- Line 36 executes the contents of `minStatement`, and the `ResultSet` reference that is returned is assigned to the `minResult` variable.
- The `if` statement in line 37 advances the `ResultSet` object's cursor, and line 38 gets the value of column 1 and assigns it to the `lowest` variable.

The code in lines 41 through 43 gets the highest price in the table. Here is a summary of how the code works:

- Line 41 executes the contents of `maxStatement`, and the `ResultSet` reference that is returned is assigned to the `maxResult` variable.
- The `if` statement in line 42 advances the `ResultSet` object's cursor, and line 43 gets the value of column 1 and assigns it to the `highest` variable.

The code in lines 46 through 48 gets the average price in the table. Here is a summary of how the code works:

- Line 46 executes the contents of `avgStatement`, and the `ResultSet` reference that is returned is assigned to the `avgResult` variable.
- The `if` statement in line 47 advances the `ResultSet` object's cursor, and line 48 gets the value of column 1 and assigns it to the `average` variable.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

22.11 What is a `ResultSet` object?

22.12 Look at the following SQL statement.

```
SELECT Id FROM Account
```

What is the name of the table from which this statement is retrieving data?

What is the name of the column that is being retrieved?

22.13 Assume that a database has a table named `Inventory`, with the following columns:

| Column Name | Type |
|---|---|
| `ProductID` | `CHAR(10)` |
| `QtyOnHand` | `INT` |
| `Cost` | `DOUBLE` |

a) Write a `SELECT` statement that will return all of the columns from every row in table.
b) Write a `SELECT` statement that will return the `ProductID` column from every row in table.
c) Write a `SELECT` statement that will return the `ProductID` column and the `QtyOnHand` column from every row in table.
d) Write a `SELECT` statement that will return the `ProductID` column only from the rows where `Cost` is less than 17.00.
e) Write a `SELECT` statement that will return all of the columns from the rows where `ProductID` ends with "ZZ".

22.14 What is the purpose of the `LIKE` operator?

22.15 What is the purpose of the `%` symbol in a character pattern used by the `LIKE` operator? What is the purpose of the underline ( _ ) character?

22.16 How can you sort the results of a `SELECT` statement on a specific column?

22.17 Assume that the following declarations exist:

```
final String DB_URL = "jdbc:derby:CoffeeDB";
String sql = "SELECT * FROM Coffee";
```

Write code that uses these `String` objects to get a database connection and execute the SQL statement. Be sure to close the connection when done.

22.18 How do you submit a `SELECT` statement to the DBMS?

22.19 Where does a `ResultSet` object's cursor initially point? How do you move the cursor forward in the result set?

22.20 Assume that a valid `ResultSet` object exists, populated with data. What method do you call to retrieve column 3 as a string? What do you pass as an argument to the method?

## 22.4 Inserting Rows

**CONCEPT: You use the `INSERT` statement in SQL to insert a new row into a table.**

In SQL, the `INSERT` statement is used to insert a row into a database table.

```
INSERT INTO  TableName  VALUES (Value1, Value2, etc...)
```

In the general format, *`TableName`* is the name of the database table. *`Value1`*, *`Value2`*, etc`...`is a list of column values. After the statement executes, a row containing the specified column values will be inserted into the table. Here is an example that inserts a row into the `Coffee` table, in our `CoffeeDB` database:

```
INSERT INTO Coffee VALUES ('Honduran Dark', '22-001', 8.65)
```

Notice that the string values are enclosed in single-quote marks. Also, notice the order that the values appear in the list. The first value, 'Honduran Dark', is inserted into the first column of the table, which is `Description`. The second value, '22-001', is inserted into the second column of the table, which is `ProdNum`. The third value, 8.65, is inserted into the third column of the table, which is `Price`. After this statement executes, a new row will be inserted into the `Coffee` table containing the following column values:

```
Description: 'Honduran Dark'
ProdNum: 22-001
Price: 8.65
```

If you are not sure of the order in which the columns appear in the table, you can use the following general format of the `INSERT` statement to specify the column names and their corresponding values.

```
INSERT INTO TableName
    (ColumnName1, ColumnName2, etc...)
VALUES
    (Value1, Value2, etc...)
```

In this general format *`ColumnName1, ColumnName2, etc...`*is a list of column names and *`Value1, Value2, etc...`*is a list of corresponding values. In the new row, *`Value1`* will appear in the column specified by *`ColumnName1, Value2`* will appear in the column specified by *`ColumnName2`*, and so forth. Here is an example:

```
INSERT INTO Coffee
    (Description, ProdNum, Price)
VALUES
    ('Honduran Dark', '22-001', 8.65)
```

This statement will produce a new row containing the following column values:

```
Description: 'Honduran Dark';
ProdNum: 22-001
Price: 8.65
```

If we rewrote the `INSERT` statement in the following manner, it would produce a new row with the same values:

```
INSERT INTO Coffee
    (ProdNum, Price, Description)
VALUES
    ('22-001', 8.65, 'Honduran Dark')
```

**NOTE:** If a column is a primary key, it must hold a unique value for each row in the table. No two rows in a table can have the same value in the primary key column. Recall that the `ProdNum` column is the primary key in the `Coffee` table. The DBMS will not allow you to insert a new row with the same product number as an existing row.

## Inserting Rows with JDBC

To issue an `INSERT` statement with JDBC, you must first get a `Statement` object from the `Connection` object, using its `createStatement` method. You then use the `Statement` object's `executeUpdate` method. The method returns an `int` value representing the number of rows that were inserted into the table. Here is an example:

```
String sqlStatement = "INSERT INTO Coffee " +
                      "(ProdNum, Price, Description) " +
                      "VALUES ('22-001', 8.65, 'Honduran Dark')";
int rows = stmt.executeUpdate(sqlStatement);
```

The first statement creates a string containing an `INSERT` statement. The second statement passes this string as an argument to the `executeUpdate` method. The method should return the `int` value 1, indicating that one row was inserted into the table. The program in Code Listing 22-7 shows an example. It prompts the user for the description, product number, and price of a new coffee and inserts that data into the `Coffee` table.

**Code Listing 22-7** **(`CoffeeInserter.java`)**

```
import java.util.Scanner;
import java.sql.*;

/**
   This program lets the user insert a row into the
   CoffeeDB database's Coffee table.
*/

public class CoffeeInserter
{
   public static void main(String[] args)
   {
      String description; // To hold the coffee description
      String prodNum;     // To hold the product number
      double price;       // To hold the price

      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Get the data for the new coffee.
         System.out.print("Enter the coffee description: ");
         description = keyboard.nextLine();
         System.out.print("Enter the product number: ");
         prodNum = keyboard.nextLine();
         System.out.print("Enter the price: ");
         price = keyboard.nextDouble();

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Create a string with an INSERT statement.
         String sqlStatement = "INSERT INTO Coffee " +
                     "(ProdNum, Price, Description) " +
                     "VALUES ('" +
                     prodNum + "', " +
                     price + ", '" +
                     description + "')";

         // Send the statement to the DBMS.
         int rows = stmt.executeUpdate(sqlStatement);

         // Display the results.
         System.out.println(rows + " row(s) added to the table.");

         // Close the connection.
         conn.close();
      }
      catch(Exception ex)
      {
         System.out.println("ERROR: " + ex.getMessage());
      }
   }
}
```

**Program Output**

```
Enter the coffee description: Honduran Dark [Enter]
Enter the product number: 22-001 [Enter]
Enter the price: 8.65 [Enter]
1 row(s) added to the table.
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

22.21 Write an SQL statement to insert a new row into the `Coffee` table containing the following data:

`Description`: Eastern Blend
`ProdNum`: 30-001
`Price`: 18.95

22.22 Rewrite the following `INSERT` statement so that it specifies the `Coffee` table's column names.

```
INSERT INTO Coffee
VALUES ('Honduran Dark', '22-001', 8.65)
```

## 22.5 Updating and Deleting Existing Rows

**CONCEPT: You use the `UPDATE` statement in SQL to change the value of an existing row. You use the `DELETE` statement to delete rows from a table.**

In SQL, the `UPDATE` statement is used to change the contents of an existing row in a table. For example, if the price of Brazilian Decaf coffee changes, we could use an `UPDATE` statement to change the `Price` column for that row. Here is the general format of the `UPDATE` statement:

```
UPDATE Table
   SET Column = Value
   WHERE Criteria
```

In the general format, *`Table`* is a table name, *`Column`* is a column name, *`Value`* is a value to store in the column, and *`Criteria`* is a conditional expression. Here is an `UPDATE` statement that will change the price of Brazilian Decaf coffee to 9.95:

```
UPDATE Coffee
   SET Price = 9.95
   WHERE Description = 'Brazilian Decaf'
```

Here is another example:

```
UPDATE Coffee
   SET Description = 'Galapagos Organic Medium'
   WHERE ProdNum = '20-001'
```

This statement locates the row where `ProdNum` is '20-001' and sets the `Description` field to `'Galapagos Organic Medium'`.

It is possible to update more than one row. For example, suppose we wish to change the price of every Guatemalan coffee to 12.95. If you look back at Table 22-1 you will see that the product number for each of the Guatemalan coffees begins with '21'. All we need is an `UPDATE` statement that locates all the rows where the `ProdNum` column begins with '21', and changes the `Price` column of those rows to 12.95. Here is such a statement:

```
UPDATE Coffee
   SET Price = 12.95
   WHERE ProdNum LIKE '21%'
```

**WARNING!** Be careful that you do not leave out the `WHERE` clause and the conditional expression when using an `UPDATE` statement. You could change the contents of every row in the table! For example, look at the following statement:

```
UPDATE Coffee
   SET Price = 4.95
```

Because this statement does not have a `WHERE` clause, it will change the `Price` column for every row in the `Coffee` table to 4.95!

## Updating Rows with JDBC

The process of issuing an `UPDATE` statement in JDBC is similar to that of issuing an `INSERT` statement. First, you get a `Statement` object from the `Connection` object, using its `createStatement` method. You then use the `Statement` object's `executeUpdate` method to issue the `UPDATE` statement. The method returns an `int` value representing the number of rows that were affected by the `UPDATE` statement. Here is an example:

```
String sqlStatement = "UPDATE Coffee " +
                      "SET Price = 9.95 " +
                      "WHERE Description = 'Brazilian Decaf'";
int rows = stmt.executeUpdate(sqlStatement);
```

The first statement creates a string containing an `UPDATE` statement. The second statement passes this string as an argument to the `executeUpdate` method. The method returns an `int` value indicating the number of rows that were changed.

Code Listing 22-8 demonstrates how to update a row in the `Coffee` table. The user enters an existing product number, and the program displays that product's data. The user then enters a new price for the specified product, and the program updates the row with the new price.

**Code Listing 22-8** (`CoffeePriceUpdater.java`)

```
import java.util.Scanner;
import java.sql.*;

/**
   This program lets the user change the price of a
   coffee in the CoffeeDB database's Coffee table.
*/

public class CoffeePriceUpdater
{
   public static void main(String[] args)
   {
      String prodNum;    // To hold the product number
      double price;      // To hold the price

      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Get the product number for the desired coffee.
         System.out.print("Enter the product number: ");
         prodNum = keyboard.nextLine();

         // Display the coffee's current data.
         if (findAndDisplayProduct(stmt, prodNum))
         {
            // Get the new price.
            System.out.print("Enter the new price: ");
            price = keyboard.nextDouble();

            // Update the coffee with the new price.
            updatePrice(stmt, prodNum, price);
         }
         else
         {
            // The specified product number was not found.
            System.out.println("That product was not found.");
         }

         // Close the connection.
         conn.close();
      }
      catch(Exception ex)
      {
         System.out.println("ERROR: " + ex.getMessage());
      }
   }

   /**
      The findAndDisplayProduct method finds a specified coffee's
      data and displays it.
      @param stmt A Statement object for the database.
      @param prodNum The product number for the desired coffee.
      @return true/false indicating whether the product was found.
   */

   public static boolean findAndDisplayProduct(Statement stmt,
                                               String prodNum)
                                               throws SQLException
   {
      boolean productFound; // Flag

      // Create a SELECT statement to get the specified
      // row from the Coffee table.
      String sqlStatement =
         "SELECT * FROM Coffee WHERE ProdNum = '" +
         prodNum + "'";

      // Send the SELECT statement to the DBMS.
      ResultSet result = stmt.executeQuery(sqlStatement);

      // Display the contents of the result set.
      if (result.next())
      {
         // Display the product.
         System.out.println("Description: " +
                            result.getString("Description"));
         System.out.println("Product Number: " +
                            result.getString("ProdNum"));
         System.out.println("Price: $" +
                            result.getDouble("Price"));

         // Set the flag to indicate the product was found.
         productFound = true;
      }
      else
      {
         // Indicate the product was not found.
         productFound = false;
      }

      return productFound;
   }

   /**
      The updatePrice method updates a specified coffee's price.
      @param stmt A Statement object for the database.
      @param prodNum The product number for the desired coffee.
      @param price The product's new price.
   */

```

```
113    public static void updatePrice(Statement stmt, String prodNum,
114                                   double price) throws SQLException
115    {
116       // Create an UPDATE statement to update the price
117       // for the specified product number.
118       String sqlStatement = "UPDATE Coffee " +
119                        "SET Price = " + Double.toString(price) +
120                        "WHERE ProdNum = '" + prodNum + "'";
121
122       // Send the UPDATE statement to the DBMS.
123       int rows = stmt.executeUpdate(sqlStatement);
124
125       // Display the results.
126       System.out.println(rows + " row(s) updated.");
127    }
128 }
```

**Program Output**

```
Enter the product number: 17-001 [Enter]
Description: Sumatra Dark
Product Number: 17-001
Price: $7.95
Enter the new price: 9.95 [Enter]
1 row(s) updated.
```

In the `main` method, line 26 gets a connection to the database and line 29 creates a `Statement` object. Lines 32 and 33 prompt the user for a product number, which is assigned to the `prodNum` variable.

Before we let the user change the specified product's price, we want to display the product's current information. So, line 36 calls a method named `findAndDisplayProduct`, passing the `Statement` object and the `prodNum` variable as arguments. The `findAndDisplayProduct` method (which is shown in lines 68 through 104) queries the database table for the row with the specified product number. If the row is found, the method displays the row's contents and then returns true. If the row is not found, the method simply returns false.

If the specified product is found, lines 39 and 40 prompt the user for the product's new price, and the user's input is assigned to the `price` variable. Then, line 43 calls a method named `updatePrice`, passing the `Statement` object, the `prodNum` variable, and the `price` variable as arguments. The `updatePrice` method (which is shown in lines 113 through 127) updates the row containing the specified product number with the new price.

Notice that neither the `findAndDisplayProduct` method nor the `updatePrice` method handles any `SQLExceptions` that might occur. If an `SQLException` happens in either of those methods, it gets passed up to the `main` method, where it is handled by the `try-catch` statement.

## Deleting Rows with the `DELETE` Statement

In SQL you use the `DELETE` statement to delete one or more rows from a table. The general format of the `DELETE` statement is:

```
DELETE FROM Table WHERE Criteria
```

In the general format, *`Table`* is a table name and *`Criteria`* is a conditional expression. Here is a `DELETE` statement that will delete the row where `ProdNum` is 20-001:

```
DELETE FROM Coffee WHERE ProdNum = '20-001'
```

This statement locates the row in the `Coffee` table where the `ProdNum` column is set to the value `'20-001'`, and deletes that row.

It is possible to delete multiple rows with the `DELETE` statement. For example, look at the following statement:

```
DELETE FROM Coffee WHERE Description LIKE 'Sumatra%'
```

This statement will delete all rows in the `Coffee` table where the `Description` column begins with ‘Sumatra’. If you glance back at Table 22-1 you will see that four rows will be deleted.

**WARNING!** Be careful that you do not leave out the `WHERE` clause and the conditional expression when using a `DELETE` statement. You could delete every row in the table! For example, look at the following statement:

```
DELETE FROM Coffee
```

Because this statement does not have a `WHERE` clause, it will delete every row in the `Coffee` table!

## Deleting Rows with JDBC

The process of issuing a `DELETE` statement in JDBC is similar to that of issuing an `INSERT` statement or an `UPDATE` statement. First, you get a `Statement` object from the `Connection` object, using its `createStatement` method. You then use the `Statement` object’s `executeUpdate` method to issue the `DELETE` statement. The method returns an `int` value representing the number of rows that were deleted. Here is an example:

```
String sqlStatement = "DELETE FROM Coffee " +
                      "WHERE ProdNum = '20-001'";
int rows = stmt.executeUpdate(sqlStatement);
```

The first statement creates a string containing a `DELETE` statement. The second statement passes this string as an argument to the `executeUpdate` method. The method returns an `int` value indicating the number of rows that were deleted.

The program shown in Code Listing 22-9 demonstrates how a row can be deleted from the `Coffee` table.

**Code Listing 22-9** (`CoffeeDeleter.java`)

```java
import java.util.Scanner;
import java.sql.*;

/**
   This program lets the user delete a coffee
   from the CoffeeDB database's Coffee table.
*/

public class CoffeeDeleter
{
   public static void main(String[] args)
   {
      String prodNum;     // To hold the product number
      String sure;        // To make sure the user wants to delete

      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Get the product number for the desired coffee.
         System.out.print("Enter the product number: ");
         prodNum = keyboard.nextLine();

         // Display the coffee's current data.
         if (findAndDisplayProduct(stmt, prodNum))
         {
            // Make sure the user wants to delete this product.
            System.out.print("Are you sure you want to delete " +
                             "this item? (y/n): ");
            sure = keyboard.nextLine();

            if (Character.toUpperCase(sure.charAt(0)) == 'Y')
            {
               // Delete the specified coffee.
               deleteCoffee(stmt, prodNum);
```

```
             }
             else
             {
                System.out.println("The item was not deleted.");
             }
          }
          else
          {
             // The specified product number was not found.
             System.out.println("That product was not found.");
          }

          // Close the connection.
          conn.close();
       }
       catch(Exception ex)
       {
          System.out.println("ERROR: " + ex.getMessage());
       }
   }

   /**
      The findAndDisplayProduct method finds a specified coffee's
      data and displays it.
      @param stmt A Statement object for the database.
      @param prodNum The product number for the desired coffee.
      @return true/false to indicate whether the product was found.
   */

   public static boolean findAndDisplayProduct(Statement stmt,
                                               String prodNum)
                                               throws SQLException
   {
      boolean productFound; // Flag

      // Create a SELECT statement to get the specified
      // row from the Coffee table.
      String sqlStatement =
         "SELECT * FROM Coffee WHERE ProdNum = '" +
         prodNum + "'";

      // Send the SELECT statement to the DBMS.
      ResultSet result = stmt.executeQuery(sqlStatement);

      // Display the contents of the result set.
      if (result.next())
      {
            // Display the product.
            System.out.println("Description: " +
                    result.getString("Description"));
            System.out.println("Product Number: " +
                    result.getString("ProdNum"));
            System.out.println("Price: $" +
                     result.getDouble("Price"));

            // Set the flag to indicate the product was found.
            productFound = true;
      }
      else
      {
         // Indicate the product was not found.
         productFound = false;
      }

      return productFound;
   }

  /**
     The deleteCoffee method deletes a specified coffee.
     @param stmt A Statement object for the database.
     @param prodNum The product number for the desired coffee.
  */

  public static void deleteCoffee(Statement stmt, String prodNum)
                                  throws SQLException
  {
     // Create a DELETE statement to delete the
     // specified product number.
     String sqlStatement = "DELETE FROM Coffee " +
                           "WHERE ProdNum = '" + prodNum + "'";

     // Send the DELETE statement to the DBMS.
     int rows = stmt.executeUpdate(sqlStatement);

     // Display the results.
     System.out.println(rows + " row(s) deleted.");
  }
}
```

**Program Output**

```
Enter the product number: 20-001 [Enter]
Description: Galapagos Medium
Product Number: 20-001
Price: $6.85
Are you sure you want to delete this item? (y/n): y [Enter]
1 row(s) deleted.
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

22.23 The Midnight Coffee Roastery is running a special on decaf coffee. Write an SQL statement that changes the price of all decaf coffees to 4.95.

22.24 The sale on decaf coffee didn't do too well, so the Midnight Coffee Roastery has decided to stop selling decaf. Write an SQL statement that will delete all decaf coffees from the `Coffee` table.

## 22.6 Creating and Deleting Tables

**CONCEPT: In SQL, the `CREATE TABLE` statement can be used to create a database table. The `DROP TABLE` statement can be used to delete a table.**

The `CoffeeDB` database that we have been using as our example is very simple. It has only one table, `Coffee`, which holds product information. The usefulness of this database is limited to looking up coffee descriptions, product numbers, and prices.

Suppose we want to store other data in the database, such as a list of customers. To do so, we would have to add another table to the database. In SQL you use the `CREATE TABLE` statement to create a table. Here is the general format of the `CREATE TABLE` statement:

```
CREATE TABLE TableName
   (ColumnName1 DataType1,
   ColumnName2 DataType2,
   etc...)
```

In the general format, *`TableName`* is the name of the table. *`ColumnName1`* is the name of the first column, and *`DataType1`* is the SQL data type for the first column. *`ColumnName2`* is the name of the second column, and *`DataType2`* is the SQL data type for the second column. This sequence repeats for all of the columns in the table. Here is an example:

```
CREATE TABLE Customer
  ( Name CHAR(25),
    Address CHAR(25),
    City CHAR(12),
    State CHAR(2),
    Zip CHAR(5) )
```

This statement creates a new table named `Customer`. The columns in the `Customer` table are `Name`, `Address`, `City`, `State`, and `Zip`.

You may also specify that a column is a primary key by listing the `PRIMARY KEY` qualifier after the column's data type. Recall from our earlier discussion on database organization that a primary key is a column that holds a unique value for each row, and can be used to identify specific rows. When you use the `PRIMARY KEY` qualifier with a column, you should also use the `NOT NULL` qualifier. The `NOT NULL` qualifier specifies that the column must contain a value for every row. Here is an example of how we can create a `Customer` table, using the `CustomerNumber` column as the primary key:

```
CREATE TABLE Customer
  ( CustomerNumber CHAR(10) NOT NULL PRIMARY KEY,
    Name CHAR(25),
    Address CHAR(25),
    City CHAR(12),
    State CHAR(2),
    Zip CHAR(5) )
```

This statement creates a new table named `Customer`. It has the same structure as the table created by the previous example, with one additional column, `CustomerNumber`, which is the primary key. Because `CustomerNumber` is the primary key, this column must hold a unique value for each row in the table.

**TIP:** Remember, a primary key is used to identify a specific row in a table. When selecting a column as a primary key, make sure it holds a unique value that cannot be duplicated for two rows in the table.

Take a look at the program in Code Listing 22-10. When you run this program, it creates the `Customer` table in the `CoffeeDB` database, and then inserts the three rows shown in Table 22-5.

**Code Listing 22-10** `(CreateCustomerTable.java)`

```
import java.sql.*;   // Needed for JDBC classes

/**
   This program creates a Customer
   table in the CoffeeDB database.
*/

public class CreateCustomerTable
{
   public static void main(String[] args)
   {
      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      try
      {
         // Create a connection to the database.
         Connection conn = DriverManager.getConnection(DB_URL);

         // Get a Statement object.
         Statement stmt = conn.createStatement();

         // Make an SQL statement to create the table.
         String sql = "CREATE TABLE Customer" +
            "( CustomerNumber CHAR(10) NOT NULL PRIMARY KEY, " +
            " Name CHAR(25)," +
            " Address CHAR(25)," +
            " City CHAR(12)," +
            " State CHAR(2)," +
            " Zip CHAR(5) )";

         // Execute the statement.
         stmt.execute(sql);

         // Add some rows to the new table.
         sql = "INSERT INTO Customer VALUES" +
               "('101', 'Downtown Cafe', '17 N. Main Street'," +
               " 'Asheville', 'NC', '55515')";
         stmt.executeUpdate(sql);

         sql = "INSERT INTO Customer VALUES" +
               "('102', 'Main Street Grocery'," +
               " '110 E. Main Street'," +
               " 'Canton', 'NC', '55555')";
         stmt.executeUpdate(sql);

         sql = "INSERT INTO Customer VALUES" +
               "('103', 'The Coffee Place', '101 Center Plaza'," +
               " 'Waynesville', 'NC', '55516')";
         stmt.executeUpdate(sql);

         // Close the connection.
         conn.close();
      }
      catch (Exception ex)
      {
         System.out.println("ERROR: " + ex.getMessage());
      }
   }
}
```

**Table 22-5** Rows inserted into the `Customer` table

| `CustomerNumber` | `Name` | `Address` | `City` | `State` | `Zip` |
|---|---|---|---|---|---|
| 101 | Downtown Cafe | 17 N. Main Street | Asheville | NC | 55515 |
| 102 | Main Street Grocery | 110 E. Main Street | Canton | NC | 55555 |
| 103 | The Coffee Place | 101 Center Plaza | Waynesville | NC | 55516 |

## Removing a Table with the `DROP TABLE` Statement

Should the need arise to delete a table from a database, you can use the `DROP TABLE` statement. Here is the statement's general format:

```
DROP TABLE TableName
```

In the general format, *`TableName`* is the name of the table you wish to delete. For example, suppose that after we created the `Customer` table, we discovered that we selected the wrong data type for many of the columns. We could delete the table, and then recreate it with the proper data types. The SQL statement to delete the table would be:

```
DROP TABLE Customer
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

22.25 Write the SQL statement to create a table named `Book`. The `Book` table should have the columns to hold the name of the publisher, the name of the author, the number of pages, and a 10-character ISBN number.

22.26 Write a statement to delete the `Book` table you created in Checkpoint 22.25.

## 22.7 Creating a New Database with JDBC

**CONCEPT: Creating a new database with JDBC is as simple as adding an attribute to the database URL and then using SQL to create a table in the database.**

In the previous section you learned about the `CREATE TABLE` statement, which is used to create a new table in an existing database. But, suppose you wish to create a completely new database. With JDBC, all you must do is append the attribute `;create=true` to the database URL. For example, suppose you wish to create a new database named `EntertainmentDB`, to hold data on your collection of DVDs. In either Java DB or Apache Derby, the URL you would use would be:

```
"jdbc:derby:EntertainmentDB;create=true"
```

Because we have appended the attribute `;create=true` to the database URL, the program will create the database when it runs. Then, we can use a `CREATE TABLE` statement to create a table in the database. The program in Code Listing 22-11 demonstrates.

**Code Listing 22-11** (`BuildEntertainmentDB.java`)

```
import java.sql.*;

/**
   This program shows how to create a new database
   using either Java DB or Apache Derby.
*/
```

```
 7
 8  public class BuildEntertainmentDB
 9  {
10     public static void main(String[] args)
11                        throws Exception
12     {
13        final String DB_URL =
14              "jdbc:derby:EntertainmentDB;create=true";
15
16        try
17        {
18           // Create a connection to the database.
19           Connection conn =
20                       DriverManager.getConnection(DB_URL);
21
22           // Create a Statement object.
23           Statement stmt = conn.createStatement();
24
25           // Create the Dvd table.
26           System.out.println("Creating the Dvd table...");
27           stmt.execute("CREATE TABLE Dvd (" +
28                        "Title CHAR(25), " +
29                        "Minutes INTEGER, " +
30                        "Price DOUBLE)");
31
32           // Close the resources.
33           stmt.close();
34           conn.close();
35           System.out.println("Done");
36        }
37        catch(Exception ex)
38        {
39           System.out.println("ERROR: " + ex.getMessage());
40        }
41     }
42  }
```

When this program runs, the `EntertainmentDB` database will be created. This is because the database URL, in line 14, has the `;create=true` attribute. Lines 27 through 30 then create a table named `Dvd`.

**NOTE:** When you create a new database using either Java DB or Apache Derby, you will see a folder appear on your system with the same name as the database. This folder holds the database. To delete the entire database, simply delete the folder.

## 22.8 Scrollable Result Sets

**CONCEPT: A scrollable result set allows random cursor movement. By default, a result set is not scrollable.**

By default, `ResultSet` objects allow you to move the cursor forward only. Once the cursor has moved past a row, you cannot move the cursor backward to read that row again. If you need to move the cursor backward through the result set, you can create a *scrollable result set*. You do this when you create a `Statement` object by using an overloaded version of a `Connection` object's `createStatement` method. The method accepts two arguments. The first specifies the result set's scrolling type. You can use any of the following constants for this argument:

```
ResultSet.TYPE_FORWARD_ONLY
```

This is the default scrolling type. It specifies that the result set's cursor should move forward only.

```
ResultSet.TYPE_SCROLL_INSENSITIVE
```

This specifies that the result set should be scrollable, allowing the cursor to move forward and backward through the result set. In addition, this result set is insensitive to changes made to the database. This means that if another program or process makes changes to the database, those changes will not appear in this result set.

```
ResultSet.TYPE_SCROLL_SENSITIVE
```

This specifies that the result set should be scrollable, allowing the cursor to move forward and backward through the result set. In addition, this result set is sensitive to changes made to the database. This means that if another program or process makes changes to the database, those changes will appear in this result set as soon as they are made.

The second argument specifies the result set's concurrency level. You can use any of the following constants for this argument:

```
ResultSet.CONCUR_READ_ONLY
```

This is the default concurrency level. It specifies that the result set contains a read-only version of data from the database. You cannot change the contents of the database by altering the contents of the result set.

```
ResultSet.CONCUR_UPDATEABLE
```

This specifies that the result set should be updateable. Changes can be made to the result set, and then those changes can be saved to the database. The `ResultSet` interface specifies several methods that may be used to update the result set and then save those updates to the database. These methods allow you to make changes to the database without issuing SQL statements. For more information on these methods, see the Java API documentation.

Assuming that `conn` references a `Connection` object, here is an example of the method call:

```
Statement stmt =
    conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_READ_ONLY);
```

The `Statement` object created by this code will be scrollable, insensitive to changes made to the database by other processes, and will not be updateable.

### `ResultSet` Navigation Methods

Once you have created a scrollable result set, you can use the following `ResultSet` methods to move the cursor:

| | |
|---|---|
| `first()` | Moves the cursor to the first row in the result set. |
| `last()` | Moves the cursor to the last row in the result set. |
| `next()` | Moves the cursor to the next row in the result set. |
| `previous()` | Moves the cursor to the previous row in the result set. |
| `relative(`*`rows`*`)` | Moves the cursor the number of rows specified by the argument *`rows`*, relative to the current row. For example, the call `relative(2)` will move the cursor 2 rows forward from the current row, and `relative(-1)` will move the cursor 1 row backward from the current row. |
| `absolute(`*`row`*`)` | Moves the cursor to the row specified by the integer *`row`*. Remember, row numbering begins at 1, so the call `absolute(1)` will move the cursor to the first row in the result set. |

> **NOTE:** Scrollable result sets are not supported by all JDBC drivers. If your driver does not support scrollable result sets, it will throw an exception when you try to use an unsupported navigation method.

The following code shows a simple, yet practical use of some of these methods:

```
resultSet.last();                   // Move to the last row
int numRows = resultSet.getRow();   // Get the current row number
resultSet.first();                  // Move back to the first row
```

This code would be useful when you need to determine the number of rows in the result set before processing any of its data. The first statement moves the cursor to the last row. The second statement calls the `ResultSet` method `getRow`, which returns the row number of the current row. The third statement then moves the cursor to the first row for subsequent processing.

## 22.9 Result Set Metadata

**CONCEPT: Result set metadata describes the contents of a result set. The metadata can be used to determine which columns were returned when a query that is not known in advance is executed.**

The term *metadata* refers to data that describes other data. A `ResultSet` object has metadata, which describes the data stored in the `ResultSet`. You can use result set metadata to determine several things about a result set, including the number of columns it contains, the names of the columns, the types of each column, and much more. Result set metadata can be very useful if you are writing an application that will submit an SQL query, and you don't know in advance what columns will be returned.

Once you have a `ResultSet` object, you can call its `getMetaData` method to get a reference to a `ResultSetMetaData` object. Assuming that `resultSet` references a `ResultSet` object, here is an example:

```
ResultSetMetaData meta = resultSet.getMetaData();
```

`ResultSetMetaData` is an interface in the `java.sql` package. It specifies numerous methods, a few of which are described in Table 22-6.

**Table 22-6** A few `ResultSetMetaData` methods

| Method | Description |
|---|---|
| `int getColumnCount()` | Returns the number of columns in the result set. |
| `String getColumnName(int col)` | Returns the name of the column specified by the integer *`col`*. The first column is column 1. |
| `String getColumnTypeName(int col)` | Returns the name of the data type of the column specified by the integer *`col`*. The first column is column 1. The data type name returned is the database-specific SQL data type. |
| `int getColumnDisplaySize(int col)` | Returns the display width, in characters, of the column specified by the integer *`col`*. The first column is column 1. |
| `String getTableName(int col)` | Returns the name of the table associated with the column specified by the integer *`col`*. The first column is column 1. |

The program in Code Listing 22-12 demonstrates how metadata can be used. It asks the user to enter a `SELECT` statement for the `CoffeeDB` database, then displays information about the result set as well as the result set's contents.

**Code Listing 22-12** (`MetaDataDemo.java`)

```
import java.sql.*;
import java.util.Scanner;

/**
   This program demonstrates result set metadata.
*/

public class MetaDataDemo
{
   public static void main(String[] args) throws Exception
   {
      // Create a named constant for the URL.
      // NOTE: This value is specific for either Java DB or Apache Derby.
      final String DB_URL = "jdbc:derby:CoffeeDB";

      try
      {
         // Create a Scanner object for keyboard input.
         Scanner keyboard = new Scanner(System.in);

         // Get a SELECT statement from the user.
         System.out.println("Enter a SELECT statement for " +
                            "the CoffeeDB database:");
         String sql = keyboard.nextLine();

         // Create a connection to the database.
         Connection conn =
                DriverManager.getConnection(DB_URL);

         // Create a Statement object.
         Statement stmt = conn.createStatement();

         // Execute the query.
         ResultSet resultSet = stmt.executeQuery(sql);

         // Get the result set metadata.
         ResultSetMetaData meta = resultSet.getMetaData();

         // Display the number of columns returned.
         System.out.println("The result set has " +
                            meta.getColumnCount() +
                            " column(s).");

         // Display the column names and types.
         for (int i = 1; i <= meta.getColumnCount(); i++)
         {
            System.out.println(meta.getColumnName(i) + ", " +
                               meta.getColumnTypeName(i));
         }

         // Display the contents of the rows.
         System.out.println("\nHere are the result set rows:");
         while (resultSet.next())
         {
            // Display a row.
            for (int i = 1; i <= meta.getColumnCount(); i++)
            {
               System.out.print(resultSet.getString(i));
            }
            System.out.println();
         }

         // Close the statement and the connection.
         stmt.close();
```

```
65          conn.close();
66       }
67       catch(Exception ex)
68       {
69          System.out.println("ERROR: " + ex.getMessage());
70       }
71    }
72 }
```

**Program Output with Example Input Shown in Bold**

```
Enter a SELECT statement for the CoffeeDB database:
SELECT * FROM Coffee WHERE Price > 10.00 [Enter]
The result set has 3 column(s).
DESCRIPTION, CHAR
PRODNUM, CHAR
PRICE, DOUBLE

Here are the result set rows:
Sumatra Organic Dark        17-004      11.95
Kona Medium                 18-001      18.45
Kona Dark                   18-002      18.45
Guatemalan Decaf            21-002      10.45
```

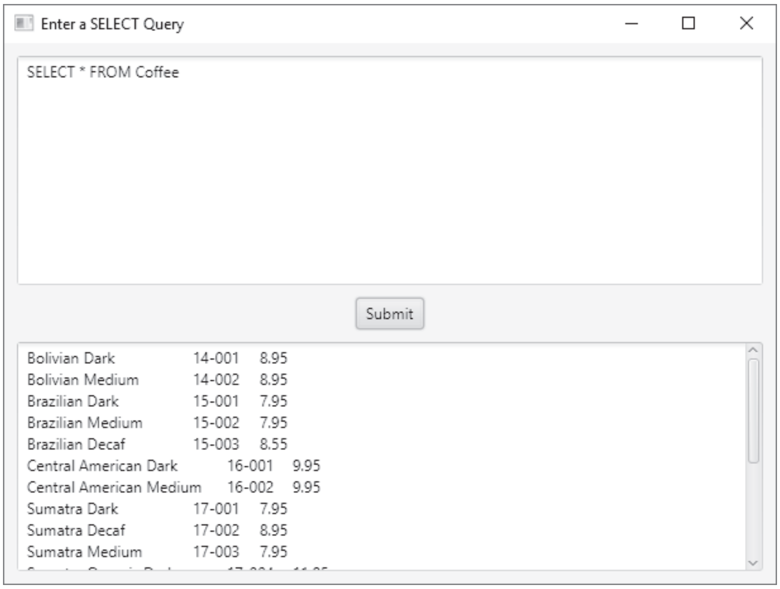
**Program Output with Example Input Shown in Bold**

```
Enter a SELECT statement for the CoffeeDB database:
SELECT ProdNum FROM Coffee WHERE Price > 10.00 [Enter]
The result set has 1 column(s).
PRODNUM, CHAR

Here are the result set rows:
17-004
18-001
18-002
21-002
```

Line 34 submits the query to the DBMS and gets a `ResultSet` object. Line 37 gets a `ResultSetMetaData` object. The statement in lines 40 through 42 displays the number of columns contained in the result set. It uses the `ResultSetMetaData` object's `getColumnCount` method to get this value. The loop in lines 45 through 49 iterates once for each column in the result set. Each iteration displays the column name and column data type. The `ResultSetMetaData` object's `getColumnName` and `getColumnTypeName` methods are used to retrieve this information. The `while` loop in lines 53 through 61 displays the contents of the result set. It has a nested `for` loop, in lines 56 through 59, which iterates once for each column in the result set. Each iteration gets the column value as a string and displays it.

Now let's look at a JavaFX GUI application that we can use as an interactive utility to view the tables in the CoffeeDB database. The DBViewer application, which is shown in Code Listing 22-13, displays the GUI shown in Figure 22-16. The user enters a `SELECT` statement in the topmost TextArea control, and then clicks the Submit button. The application executes the query and displays the results in the bottom TextArea.

**Figure 22-16** The DBViewer application (Oracle Corporate Counsel)



```
SELECT * FROM Coffee
```

```
Bolivian Dark           14-001   8.95
Bolivian Medium         14-002   8.95
Brazilian Dark          15-001   7.95
Brazilian Medium        15-002   7.95
Brazilian Decaf         15-003   8.55
Central American Dark        16-001   9.95
Central American Medium      16-002   9.95
Sumatra Dark            17-001   7.95
Sumatra Decaf           17-002   8.95
Sumatra Medium          17-003   7.95
```

**Code Listing 22-13** (`DBViewer.java`)

```java
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
import java.sql.*;

public class DBViewer extends Application
{
   // Database URL.
   final String DB_URL = "jdbc:derby:CoffeeDB";

   // Controls for input and output
   TextArea queryTextArea;
   TextArea resultsTextArea;

   public static void main(String[] args)
   {
      // Launch the application
      launch(args);
   }

```

```
@Override
public void start(Stage primaryStage)
{
   final int COL_SIZE = 50;
   final int ROW_SIZE = 10;
   final double SPACING = 10.0;

   // Build the query entry area.
   queryTextArea = new TextArea();
   queryTextArea.setPrefColumnCount(COL_SIZE);
   queryTextArea.setPrefRowCount(ROW_SIZE);

   // Build the query results area.
   resultsTextArea = new TextArea();
   resultsTextArea.setPrefColumnCount(COL_SIZE);
   resultsTextArea.setPrefRowCount(ROW_SIZE);

   // Create the Submit button.
   Button submitButton = new Button("Submit");
   submitButton.setOnAction(new SubmitButtonHandler());

   // Put the controls in a VBox.
   VBox vbox = new VBox(SPACING, queryTextArea,
                        submitButton, resultsTextArea);
   vbox.setAlignment(Pos.CENTER);
   vbox.setPadding(new Insets(SPACING));

   // Set the title.
   primaryStage.setTitle("Enter a SELECT Query");

   // Create a scene and set it to the stage.
   Scene scene = new Scene(vbox);
   primaryStage.setScene(scene);

   // Show the stage.
   primaryStage.show();
}

/**
  Event handler class for submitButton
*/

class SubmitButtonHandler implements EventHandler<ActionEvent>]
{
   @Override
   public void handle(ActionEvent event)
   {
      try
```

```
         {
            // Clear the previous results.
            resultsTextArea.setText("");

            // Create a connection to the database.
            Connection conn = DriverManager.getConnection(DB_URL);

            // Create a Statement object.
            Statement stmt = conn.createStatement();

            // Execute the query.
            ResultSet resultSet =
                      stmt.executeQuery(queryTextArea.getText());

            // Get the result set meta data.
            ResultSetMetaData meta = resultSet.getMetaData();

            // Create a string to hold the results
            String output = "";

            // Get the results.
            while (resultSet.next())
            {
               // Get all of the columns in a row.
               for (int i = 1; i <= meta.getColumnCount(); i++)
               {
                  output += resultSet.getString(i);
                  output += '\t';
               }
               output += '\n';
            }

            // Display the results.
            resultsTextArea.setText(output);

            // Close the statement and the connection.
            stmt.close();
            conn.close();

         }
         catch (SQLException e)
         {
            e.printStackTrace();
            System.exit(0);
         }
      }
   }
}
```

## 22.10 Relational Data

**CONCEPT: In a relational database, a column from one table can be associated with a column from other tables. This association creates a relationship between the tables.**

In the section 22.6 we added a `Customer` table to the `CoffeeDB` database (see Code Listing 22-10). This made the database more useful by giving us the ability to look up customer information, as well as the product information held in the `Coffee` table.

Suppose we want to make the database even more useful by storing information about unpaid customer orders. That way, we can get a list of all the customers with outstanding balances. To do this, we will need to add an additional table and more data to the database. Here is a summary of the `UnpaidOrder` table, which we will create to hold order data. (We will explain what a foreign key is momentarily.)

`UnpaidOrder` table:

| | | |
|---|---|---|
| `CustomerNumber` | `CHAR(10)` | *Foreign Key* |
| `ProdNum` | `CHAR(10)` | *Foreign Key* |
| `OrderDate` | `CHAR(10)` | |
| `Quantity` | `DOUBLE` | |
| `Cost` | `DOUBLE` | |

The first column, `CustomerNumber`, identifies the customer that placed the order. Notice, however, that the Unpaid`Order` table does not hold any other customer data. It holds only the customer number. When designing a database, it is important that you avoid unnecessary duplication of data. Because the customer data is already stored in the `Customer` table, we need only to store the customer number in the `Order` table. We can use that number to look up the rest of the customer's data in the `Customer` table.

In the `UnpaidOrder` table, the `CustomerNumber` column is considered a foreign key. A *foreign key* is a column in one table that references a primary key in another table. Recall that `CustomerNumber` is the primary key in the `Customer` table. When we add a row to the `UnpaidOrder` table, the value that we store in the `CustomerNumber` column must match a value in the `CustomerNumber` column of the `Customer` table. This creates a relationship between the rows in the `UnpaidOrder` table and the rows in the `Customer` table.

The next column, `ProdNum`, is also a foreign key because it identifies a product in the `Coffee` table. This is the item that the customer ordered. Once again, it is not necessary to store all of the product data in the `UnpaidOrder` table. We need only to store the product number, and then we can use that number to look up the product data in the `Coffee` table.

The next column in the `UnpaidOrder` table is `OrderDate`. This will hold the date that the order was placed.[1] The `Quantity` column holds the number of pounds of coffee that the customer ordered. The `Cost` column holds the total cost of the item. We will use the following SQL statement to create the `Order` table:

---

[1] In SQL there is a `DATE` data type, which is used to hold dates. It corresponds to the `java.sql.Date` class. To keep the example simple, however, we will merely store the invoice date as a string.

```
CREATE TABLE UnpaidOrder
( CustomerNumber CHAR(10) NOT NULL REFERENCES Customer(CustomerNumber),
  ProdNum CHAR(10) NOT NULL REFERENCES Coffee(ProdNum),
  OrderDate CHAR(10),
  Quantity DOUBLE,
  Cost DOUBLE )
```

Notice that this statement introduces a new qualifier, `REFERENCES`, which is used with both the `CustomerNumber` and `ProdNum` columns. Here is the way it is used with the `CustomerNumber` column:

```
REFERENCES Customer(CustomerNumber)
```

This indicates that the column references the `CustomerNumber` column in the `Customer` table. Because of this qualifier, the DBMS performs a check when you insert a row into the `UnpaidOrder` table. It will allow you to insert a row only if the `CustomerNumber` column contains a valid value from the `CustomerNumber` column of the `Customer` table. This ensures *referential integrity* between the two tables.

The `REFERENCES` qualifier is also used with the `ProdNum` column:

```
REFERENCES Coffee(ProdNum)
```

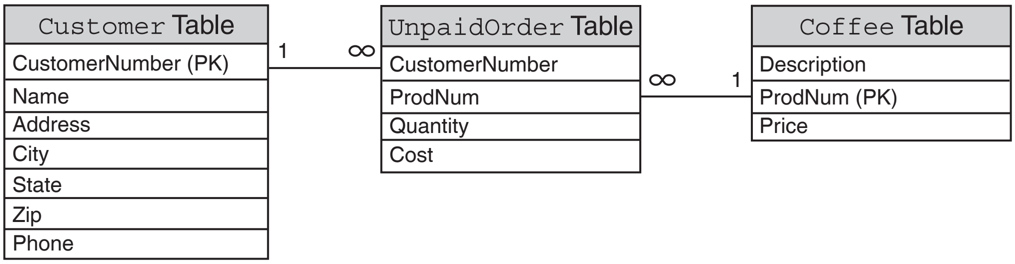
This indicates that the column references the `ProdNum` column in the `Coffee` table. When you insert a row into the `Order` table, its `ProdNum` column must contain a valid value from the `ProdNum` column of the `Coffee` table.

System designers commonly use *entity relationship diagrams* to show the relationships between database tables. Figure 22-17 shows an entity relationship diagram for the `CoffeeDB` database. In the diagram, the primary keys are denoted with (PK). The lines that are drawn between the tables show how the tables are related. In this diagram, there are two types of relationships:

- A *one to many relationship* means that for each row in table *A* there can be many rows in table *B* that reference it.
- A *many to one relationship* means that many rows in table *A* can reference a single row in table *B*.

Notice that the ends of each line show either a 1 or an infinity symbol [∞] You can interpret the infinity symbol as meaning *many*, and the number 1 as meaning *one*. Look at the line

**Figure 22-17** Entity relationship diagram



that connects the `Customer` table to the `UnpaidOrder` table. The 1 is at the end of the line near the `Customer` table, and the infinity symbol is at the end near the `UnpaidOrder` table. This means that one row in the `Customer` table may be referenced by many rows in the `UnpaidOrder` table. This makes sense because a customer can place many orders. (In fact, this is what management hopes for!)

If we look at the relationship in the other direction, we see that many of the rows in the `UnpaidOrder` table can reference one row in the `Customer` table. Here is a summary of all the relationships shown in the diagram:

- There is a one to many relationship between the `Customer` table and the `UnpaidOrder` table. One row in the `Customer` table may be referenced by many rows in the `UnpaidOrder` table.
- There is a many to one relationship between the `UnpaidOrder` table and the `Customer` table. Many rows in the `UnpaidOrder` table may reference a single row in the `Customer` table.
- There is a one to many relationship between the `Coffee` table and the `UnpaidOrder` table. One row in the `Coffee` table may be referenced by many rows in the `UnpaidOrder` table.
- There is a many to one relationship between the `UnpaidOrder` table and the `Coffee` table. Many rows in the `UnpaidOrder` table may reference a single row in the `Coffee` table.

## Joining Data from Multiple Tables

When related data is stored in multiple tables, as in the `CoffeeDB` database, it is often necessary to pull data from different tables in a `SELECT` statement. For example, suppose we want to see information about all the unpaid orders. Specifically, for each unpaid order, we want to see the customer number, customer name, order date, coffee description, and cost. This involves columns from the `Customer`, `UnpaidOrder`, and `Coffee` tables. Because some of these tables have columns with the same name, we have to use qualified column names in our `SELECT` statement. A *qualified column name* takes the following form:

```
TableName.ColumnName
```

For example, `Customer.CustomerNumber` specifies the `CustomerNumber` column in the `Customer` table, and `UnpaidOrder.CustomerNumber` specifies the `CustomerNumber` column in the `UnpaidOrder` table. Take a look at the following query:

```
SELECT
   Customer.CustomerNumber,
   Customer.Name,
   UnpaidOrder.OrderDate,
   Coffee.Description,
   UnpaidOrder.Cost
FROM
   Customer, UnpaidOrder, Coffee
```

```
WHERE
   UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
   UnpaidOrder.ProdNum = Coffee.ProdNum
```

The first part of the query specifies the columns we want:

```
SELECT
   Customer.CustomerNumber,
   Customer.Name,
   UnpaidOrder.OrderDate,
   Coffee.Description,
   UnpaidOrder.Cost
```

The second part of the query, which uses the `FROM` clause, specifies the tables we want to pull the data from:

```
FROM
   Customer, UnpaidOrder, Coffee
```

Notice that the table names are separated by commas. The third part of the query, which uses the `WHERE` clause, specifies search criteria:

```
WHERE
   UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
   UnpaidOrder.ProdNum = Coffee.ProdNum
```

The search criteria tell the DBMS how to link the rows in the tables. Recall from our earlier discussion that the `UnpaidOrder.CustomerNumber` column references the `Customer.CustomerNumber` column, and the `UnpaidOrder.ProdNum` column references the `Coffee.ProdNum` column.

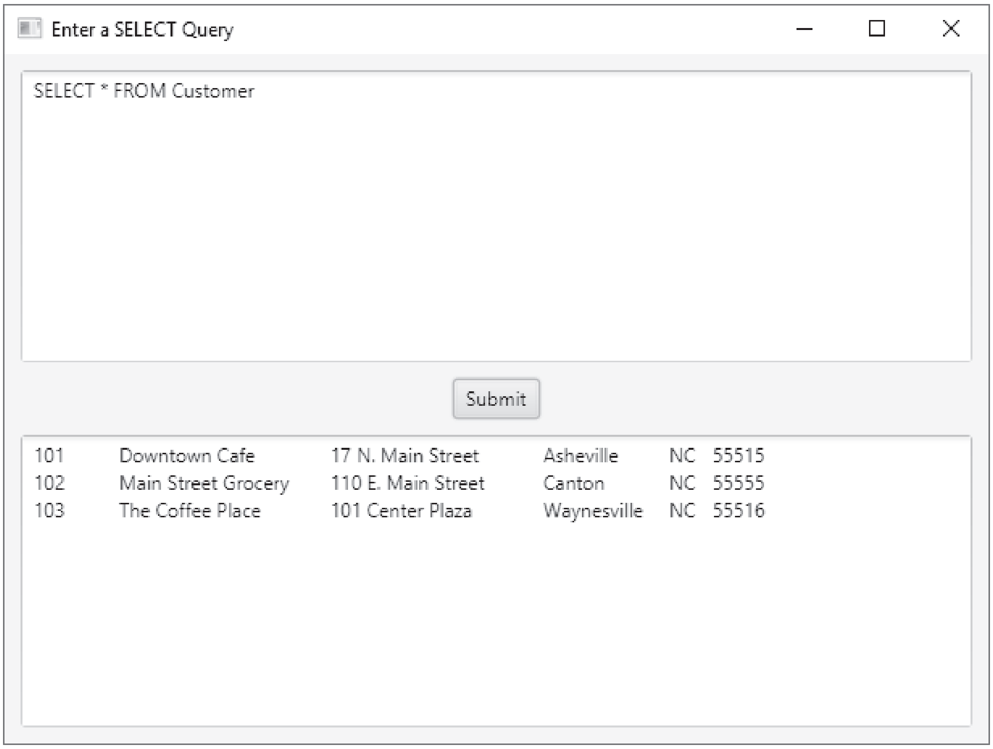
**WARNING!** When joining data from multiple tables, be sure to use a `WHERE` clause to specify search criteria that link the appropriate columns. Failure to do so will result in a large set of unrelated data.

## An Order Entry System

Now, let's look at an example application that uses a relational database. In order to use this application, you will need the `Coffee` table, the `Customer` table, and the `UnpaidOrder` table in the `CoffeeDB` database. Back in Section 22.6 you saw Code Listing 22-10, which is a program named `CreateCustomerTable.java`. This program created the `Customer` table in the `CoffeeDB` database, and added three sample rows. If you haven't already run that program, do so now. After running the program, you can run the `DBViewer` application presented earlier in this chapter, and enter the statement `SELECT * FROM Customer`. You should see the data shown in Figure 22-18.

Next, you should run the program `CreateUnpaidOrderTable.java`. This program is in the source code folder for this chapter, and it will create the `UnpaidOrder` table we discussed earlier in this chapter. The table will have no data stored in it, however.

**Figure 22-18** Customer table displayed in the `DBViewer` application (Oracle Corporate Counsel)



```
SELECT * FROM Customer
```

```
101    Downtown Cafe         17 N. Main Street    Asheville     NC   55515
102    Main Street Grocery   110 E. Main Street   Canton        NC   55555
103    The Coffee Place      101 Center Plaza     Waynesville   NC   55516
```

Now that we have the necessary tables set up in our database, we will examine an order entry application that allows the user to place an order for coffee. The application is built from two classes: one to manage the database, and another to display the GUI. The first class we will look at is the `CoffeeDBManager` class, shown in Code Listing 22-14. This class performs a variety of operations on the `CoffeeDB` database, which we will need in our order entry system.

**Code Listing 22-14** (`CoffeeDBManager.java`)

```
import java.util.ArrayList;
import java.sql.*;

/**
   The CoffeeDBManager class performs operations
   the CoffeeDB database.
*/

public class CoffeeDBManager
{
   // Constant for database URL.
   public final static String DB_URL = "jdbc:derby:CoffeeDB";

  /**
      The getCustomerNames method returns an ArrayList
```

```
    of Strings containing all the customer names.
*/
public static ArrayList<String> getCustomerNames() throws SQLException
{
    // Create a connection to the database.
    Connection conn = DriverManager.getConnection(DB_URL);

    // Create a connection to the database.
    conn = DriverManager.getConnection(DB_URL);

    // Create a Statement object for the query.
    Statement stmt =
       conn.createStatement(
               ResultSet.TYPE_SCROLL_SENSITIVE,
               ResultSet.CONCUR_READ_ONLY);

    // Execute the query.
    ResultSet resultSet = stmt.executeQuery(
                        "SELECT Name FROM Customer");

    // Get the number of rows
    resultSet.last();                  // Move last row
    int numRows = resultSet.getRow();  // Get row number
    resultSet.first();                 // Move to first row

    // Create an ArrayList for the customer names.
    ArrayList<String> listData = new ArrayList<>();

    // Populate the ArrayList with customer names.
    for (int index = 0; index < numRows; index++)
    {
       // Store the customer name in the array.
       listData.add(resultSet.getString(1));

       // Go to the next row in the result set.
       resultSet.next();
    }

    // Close the connection and statement objects.
    conn.close();
    stmt.close();

    // Return the listData array.
    return listData;
}

/**
    The getCoffeeNames method returns an array
```

```
       of Strings containing all the coffee names.
   */
   public static ArrayList<String> getCoffeeNames() throws SQLException
   {
       // Create a connection to the database.
       Connection conn = DriverManager.getConnection(DB_URL);

       // Create a Statement object for the query.
       Statement stmt =
          conn.createStatement(
                  ResultSet.TYPE_SCROLL_SENSITIVE,
                  ResultSet.CONCUR_READ_ONLY);

       // Execute the query.
       ResultSet resultSet = stmt.executeQuery(
                        "SELECT Description FROM Coffee");

       // Get the number of rows
       resultSet.last();                   // Move to last row
       int numRows = resultSet.getRow();   // Get row number
       resultSet.first();                  // Move to first row

       // Create an array for the coffee names.
       ArrayList<String> listData = new ArrayList<>();

       // Populate the array with coffee names.
       for (int index = 0; index < numRows; index++)
       {
          // Store the coffee name in the array.
          listData.add(resultSet.getString(1));

          // Go to the next row in the result set.
          resultSet.next();
       }

       // Close the connection and statement objects.
       conn.close();
       stmt.close();

       // Return the listData array.
       return listData;
   }

   /**
       The getProdNum method returns a specific
       coffee's product number.
  */
   public static String getProdNum(String coffeeName) throws SQLException
   {
      String prodNum = ""; // Product number

      // Create a connection to the database.
      Connection conn = DriverManager.getConnection(DB_URL);

      // Create a Statement object for the query.
      Statement stmt = conn.createStatement();

      // Execute the query.
      ResultSet resultSet = stmt.executeQuery(
                         "SELECT ProdNum " +
                         "FROM Coffee " +
                         "WHERE Description = '" +
                         coffeeName + "'");

      // If the result set has a row, go to it
      // and retrieve the product number.
      if (resultSet.next())
         prodNum = resultSet.getString(1);

      // Close the Connection and Statement objects.
      conn.close();
      stmt.close();

      // Return the product number.
      return prodNum;
   }

   /**
      The getCoffeePrice method returns the price
      of a coffee.
   */
   public static double getCoffeePrice(String prodNum) throws SQLException
   {
      double price = 0.0; // Coffee price

      // Create a connection to the database.
      Connection conn = DriverManager.getConnection(DB_URL);

      // Create a Statement object for the query.
      Statement stmt = conn.createStatement();

      // Execute the query.
      ResultSet resultSet = stmt.executeQuery(
                         "SELECT Price " +
                         "FROM Coffee " +
                         "WHERE ProdNum = '" +
                          prodNum + ""');

      // If the result set has a row, go to it
      // and retrieve the price.
      if (resultSet.next())
         price = resultSet.getDouble(1);

      // Close the connection and statement objects.
      conn.close();
      stmt.close();

      // Return the price.
      return price;
   }

   /**
     The getCustomerNum method returns a specific
     customer's number.
   */
   public static String getCustomerNum(String name) throws SQLException
   {
      String customerNumber = "";

      // Create a connection to the database.
      Connection conn = DriverManager.getConnection(DB_URL);

      // Create a Statement object for the query.
      Statement stmt = conn.createStatement();

      // Execute the query.
      ResultSet resultSet = stmt.executeQuery(
                          "SELECT CustomerNumber " +
                          "FROM Customer " +
                          "WHERE Name = '" + name + "'");

      if (resultSet.next())
      customerNumber = resultSet.getString(1);

      // Close the connection and statement objects.
      conn.close();
      stmt.close();

      // Return the customer number.
      return customerNumber;
   }

   /**
      The submitOrder method submits an order to
```

```
208          the UnpaidOrder table in the CoffeeDB database.
209       */
210       public static void submitOrder(String custNum, String prodNum,
211                                      int quantity, double price,
212                                      String orderDate) throws SQLException
213       {
214          // Calculate the cost of the order.
215          double cost = quantity * price;
216
217          // Create a connection to the database.
218          Connection conn = DriverManager.getConnection(DB_URL);
219
220          // Create a Statement object for the query.
221          Statement stmt = conn.createStatement();
222
223          // Execute the query.
224          stmt.executeUpdate("INSERT INTO UnpaidOrder VALUES('" +
225                             custNum + "', '" +
226                             prodNum + "', '" + orderDate + "', " +
227                             quantity + ", " + cost + ")");
228
229          // Close the connection and statement objects.
230          conn.close();
231          stmt.close();
232       }
233    }
```

Here is a summary of the methods in the `CoffeeDBManager` class:

- The `getCustomerNames` method, in lines 18 through 60, returns an `ArrayList` of strings containing the names of all the customers in the `Customer` table.
- The `getCoffeeNames` method, in lines 66 through 105, returns an `ArrayList` of strings containing the names of all the coffees in the `Coffee` table.
- The `getProdNum` method, in lines 111 through 139, accepts a `String` argument containing the name of a coffee. The method returns the coffee's product number.
- The `getCoffeePrice` method, in lines 145 through 173, accepts a `String` argument containing a coffee's product number. The method returns the price of the specified coffee.
- The `getCustomerNum` method, in lines 179 through 204, accepts a `String` argument containing a customer's name. The method returns that customer's customer number.
- The `submitOrder` method, in lines 210 through 232, creates a row in the `UnpaidOrder` table. It accepts arguments for the customer number, the product number of the coffee being ordered, the quantity being ordered, the coffee's price per pound, and the order date. Line 215 calculates the cost of the order by multiplying the quantity by the price per pound. Line 218 opens a connection to the database, and line 221 creates a `Statement` object. Lines 224 through 227 execute an `INSERT` statement on the `UnpaidOrders` table.

Next, we will look at the `OrderEntrySystem` class, shown in Code Listing 22-15. This class builds the GUI and interacts with the user. When you run the application, it displays the GUI shown in Figure 22-17. To place an order, you select a customer and a coffee from the ListView controls, enter the quantity and the date in the TextField controls, and click the Submit button. The system then inserts a new row for the order into the `UnpaidOrder` table.

**Code Listing 22-15** `(OrderEntrySystem.java)`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import java.sql.SQLException;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;

public class OrderEntrySystem extends Application
{
   // Fields for input
   private ListView<String> customerListView;
   private ListView<String> coffeeListView;
   private TextField qtyTextField;
   private TextField orderDateTextField;

   // Constants
   final int LV_WIDTH = 150;
   final int LV_HEIGHT = 75;
   final double SPACING = 10.0;

   private void buildGUI(Stage stage)
   {
      // Build the customer selection area.
      Label customerPrompt = new Label("Select a Customer");
      customerListView = new ListView<String>();
      customerListView.setPrefSize(LV_WIDTH, LV_HEIGHT);
      VBox customerVBox = new VBox(SPACING, customerPrompt,
                                   customerListView);
      customerVBox.setAlignment(Pos.CENTER);
      customerVBox.setPadding(new Insets(SPACING));

      // Build the coffee selection area
      Label coffeePrompt = new Label("Select a Coffee");
      coffeeListView = new ListView<String>();
      coffeeListView.setPrefSize(LV_WIDTH, LV_HEIGHT);
      VBox coffeeVBox = new VBox(10, coffeePrompt,
                                 coffeeListView);
      coffeeVBox.setAlignment(Pos.CENTER);
      coffeeVBox.setPadding(new Insets(SPACING));

      // Build the quantity and date entry areas.
      Label qtyPrompt = new Label("Quantity");
      qtyTextField = new TextField();

      Label datePrompt = new Label("Order Date");
      orderDateTextField = new TextField();
      VBox orderVBox = new VBox(SPACING, qtyPrompt, qtyTextField,
                                datePrompt, orderDateTextField);
      orderVBox.setAlignment(Pos.CENTER);
      orderVBox.setPadding(new Insets(SPACING));

      // Build the button area.
      Button submitButton = new Button("Submit");
      Button exitButton = new Button("Exit");
      HBox buttonHBox = new HBox(SPACING, submitButton, exitButton);
      buttonHBox.setAlignment(Pos.CENTER);
      buttonHBox.setPadding(new Insets(SPACING));

      // Register event handlers for the Button controls.
      submitButton.setOnAction(new SubmitButtonHandler());
      exitButton.setOnAction(e ->
      {
          // Close the stage, end the application.
          stage.close();
      });

      // Put everything inside a BorderPane.
      BorderPane borderPane = new BorderPane();
      borderPane.setLeft(customerVBox);
      borderPane.setCenter(coffeeVBox);
      borderPane.setRight(orderVBox);
      borderPane.setBottom(buttonHBox);

      // Set the title bar text.
      stage.setTitle("Order Entry System");

      // Create a scene and add it to the stage
      Scene scene = new Scene(borderPane);
      stage.setScene(scene);
```

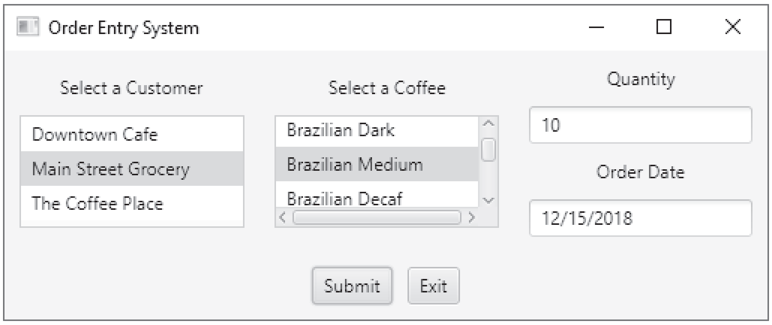
```
   }

   /**
      The loadData method loads customer names and
      coffee names into the ListView controls.
   */

   private void loadData()
   {
      try
      {
         // Load customer names into the ListView.
         customerListView.getItems().setAll(
                 CoffeeDBManager.getCustomerNames());

         // Load coffee names into the ListView.
         coffeeListView.getItems().setAll(
                 CoffeeDBManager.getCoffeeNames());
      }
      catch (SQLException e)
      {
         e.printStackTrace();
         System.exit(0);
      }
   }

   public static void main(String[] args)
   {
      // Launch the application
      launch(args);
   }

   @Override
   public void start(Stage primaryStage)
   {
      // Build the GUI.
      buildGUI(primaryStage);

      // Load data from the database.
      loadData();

      // Show the stage.
      primaryStage.show();
   }

   /**
     Event handler class for submitButton
   */

   class SubmitButtonHandler implements EventHandler<Actionevent>
   {
      @Override
      public void handle(ActionEvent event)
      {
         try
         {
            // Get the customer name from the ListView
            String customerName =
               customerListView.getSelectionModel().getSelectedItem();

            // Get the coffee name from the ListView
            String coffeeName =
               coffeeListView.getSelectionModel().getSelectedItem();

            // Get the quantity
            int qty = Integer.parseInt(qtyTextField.getText());

            // Get the order date
            String orderDate = orderDateTextField.getText();

            // Get the customer number from the DB
            String customerNum =
               CoffeeDBManager.getCustomerNum(customerName);

            // Get the coffee product number from the DB
            String prodNum = CoffeeDBManager.getProdNum(coffeeName);

            // Get the price per pound from the DB
            double price = CoffeeDBManager.getCoffeePrice(prodNum);

            // Submit the order to the DB
            CoffeeDBManager.submitOrder(customerNum, prodNum, qty,
                                        price, orderDate);

            // Clear the text fields
            qtyTextField.clear();
            orderDateTextField.clear();
         }
         catch (SQLException e)
         {
            e.printStackTrace();
            System.exit(0);
         }
      }
   }
}
```

Let's take a closer look at the code.

- Lines 20 through 23 declare the ListView and TextField fields that will be used for input.
- Lines 26 through 28 declare constants for the ListView width and height, and for the spacing that we will place around controls in our layout containers.
- The `buildGUI` method in lines 30 through 89 creates all of the controls, places them in layout containers, and registers event handlers for the buttons. A `Scene` is created, and it is set to the stage.
- The `loadData` method, in lines 96 through 113, uses the `CoffeeDBManager` class to get the customer names and coffee names from the database and load them into the ListView controls.
- The `start` method, in lines 122 through 132, calls the `buildGUI` method (line 125), calls the `loadData` method (line 128), and shows the stage (line 131).
- The `SubmitButtonHandler` class, in lines 138 through 183, is the event handler for the `submitButton` control. It's `handle` method performs the following steps:
  - It gets the selected customer name from the `customerListView` control in lines 146 and 147.
  - It gets the selected coffee name from the `coffeeListView` control in lines 150 and 151.
  - It gets the quantity from the `qtyTextField` control in line 154.
  - It gets the order date from the `orderDateTextField` in line 157.
  - At this point, we have the name of the customer placing the order and the name of the coffee being ordered, but to submit an order, we need the customer number and the product number. Lines 160 and 161 call the `CoffeeDBManager` class's `getCustomerNum` method to retrieve the customer number from the database. Line 164 calls the `CoffeeDBManager` class's `getProdNum` method to retrieve the product number. We also need the price of the coffee. Line 167 calls the `CoffeeDBManager` class's `getCoffeePrice` method to retrieve this information.
  - Lines 170 and 171 call the `CoffeeDBManager` class's `submitOrder` method to submit the order. After the order is submitted, lines 174 and 175 clear the text fields holding the quantity and order date, making it easier to enter the next order.

Figure 22-19 shows the order entry GUI with a customer selected, a coffee selected, a quantity entered, and an order date entered.

**Figure 22-19** Order data entered (Oracle Corporate Counsel)



Downtown Cafe
Main Street Grocery
The Coffee Place

Brazilian Dark
Brazilian Medium
Brazilian Decaf

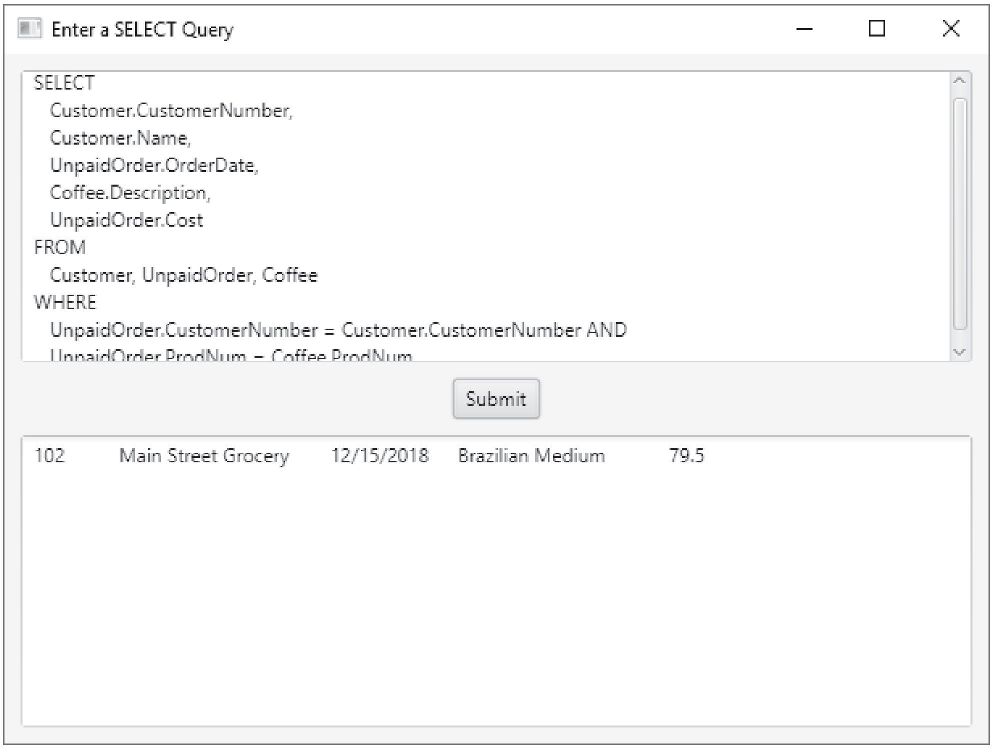
Quantity: 10

Order Date: 12/15/2018

After we submit the order shown in Figure 22-19, we can run the `DBViewer` application and enter the following `SELECT` statement to pull data from various tables relating to the order. Figure 22-20 shows the `DBViewer` application's screen with the `SELECT` statement already filled in and the results of the statement.

```
SELECT
   Customer.CustomerNumber,
   Customer.Name,
   UnpaidOrder.OrderDate,
   Coffee.Description,
   UnpaidOrder.Cost
FROM
   Customer, UnpaidOrder, Coffee
WHERE
   UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
   UnpaidOrder.ProdNum = Coffee.ProdNum
```

**Figure 22-20** Order information viewed in `DBViewer` (Oracle Corporate Counsel)



```
SELECT
  Customer.CustomerNumber,
  Customer.Name,
  UnpaidOrder.OrderDate,
  Coffee.Description,
  UnpaidOrder.Cost
FROM
  Customer, UnpaidOrder, Coffee
WHERE
  UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
  UnpaidOrder.ProdNum = Coffee.ProdNum
```

```
102    Main Street Grocery    12/15/2018    Brazilian Medium    79.5
```

# 22.11 Advanced Topics

## Transactions

Sometimes an application must perform several database updates to carry out a single task. For example, suppose you have a checking account and a car loan at your bank. Each month, your car payments are automatically taken from your checking account. For this operation to take place, the balance of your checking account must be reduced by the amount of the car payment, and the balance of the car loan must also be reduced.

An operation that requires multiple database updates is known as a *transaction*. In order for a transaction to be complete, all of the steps involved in the transaction must be performed. If any single step within a transaction fails, then none of the steps in the transaction should be performed. For example, imagine that the bank system has begun the process of making your car payment. The amount of the payment is subtracted from your checking account balance, but then some sort of system failure occurs before the balance of the car loan is reduced. You would be quite upset to learn that the amount for your car payment was withdrawn from your checking account, but never applied to your loan!

Most database systems provide a means for undoing the partially completed steps in a transaction when a failure occurs. When you write transaction-processing code, there are two concepts you must understand: commit and rollback. The term *commit* refers to making a permanent change to a database, and the term *rollback* refers to undoing changes to a database.

By default, the JDBC `Connection` class operates in auto commit mode. In *auto commit* mode, all updates that are made to the database are made permanent as soon as they are executed. When auto commit mode is turned off, however, changes do not become permanent until a commit command is executed. This makes it possible to use a rollback command to undo changes. A rollback command will undo all database changes since the last commit command.

In JDBC, you turn auto commit mode off with the `Connection` class's `setAutoCommit` method, passing the argument `false`. Here is an example:

```
conn.setAutoCommit(false);
```

You execute a commit command by calling the `Connection` class's commit method, as shown here:

```
conn.commit();
```

A rollback command can be executed with the `Connection` class's rollback method, as shown here:

```
conn.rollback();
```

Let's look at an example. Suppose we add a new table named `Inventory` to the `CoffeeDB` database, for the purpose of storing the quantity of each type of coffee in inventory. The table has two rows: `ProdNum`, which is a coffee product number, and `Qty`, which is the quantity of each type of coffee. When an order is placed, we want to update both the `Inventory` table and the `UnpaidOrder` table. In the `Inventory` table we want to subtract the quantity being ordered from the quantity in inventory. In the `UnpaidOrder` table we want to insert a new row representing the order. Here is some example code that might be used to process this as a transaction.

```
Connection conn = DriverManager.getConnection(DB_URL);
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
// Attempt the transaction.
try
{
   // Update the inventory records.
   stmt.executeUpdate("UPDATE Inventory SET Qty = Qty - " +
               qtyOrdered + " WHERE ProdNum = " + prodNum);
   // Add the order to the UnpaidOrder table.
   stmt.executeUpdate("INSERT INTO UnpaidOrder VALUES('" +
                      custNum + "', '" +
                      prodNum + "', '" + orderDate + "', " +
                      qtyOrdered + ", " + cost + ")");
   // Commit all these updates.
   conn.commit();
}
catch (SQLException ex)
{
   // Roll back the changes.
   conn.rollback();
}
```

Notice that inside the try block, after the statements to update the database have been executed, the `Connection` class's `commit` method is executed. In the catch block, the `rollback` method is executed in the event of an error.

## Stored Procedures

Many commercial database systems allow you to create SQL statements and store them in the DBMS itself. These SQL statements are called *stored procedures*, and they can be executed by other applications using the DBMS. If you have written an SQL statement that is used often in a variety of applications, it might be helpful to store it as a stored procedure in the DBMS. Then, you can call the stored procedure from any of your applications when you need to execute the SQL statement. Because stored procedures are already in the DBMS, they usually execute faster than SQL statements that are submitted from applications outside the DBMS.

We won't go into the details of stored procedures in this book, but we will point you in the right direction if you want to learn more. Each DBMS has its own syntax for creating a stored procedure in SQL, so you will have to consult your DBMS documentation to determine the format. Once you have properly written a stored procedure in SQL, you simply submit it to the DBMS using the `Statement` class's `execute` method. To execute a stored procedure, you must create a `CallableStatement` object. `CallableStatement` is an interface in the `java.sql` package. To create a `CallableStatement` object, you call the `Connection` class's `prepareCall` statement.

## 22.12 Common Errors to Avoid

- **Using the `==` operator instead of the `=` operator in an SQL statement.** The equal-to operator in SQL is one `=` sign, instead of two.
- **Using double quotes around strings instead of single quotes.** String literals in SQL are enclosed in single quotes instead of double quotes.
- **Using `&&` and `||` in an SQL statement.** The logical AND and logical OR operators in SQL are the words `AND` and `OR`, not the `&&` and `||` symbols.
- **Not using the correct `WHERE` clause in an `UPDATE` statement.** Be careful that you do not leave out the `WHERE` clause and the conditional expression when using an `UPDATE` statement. You could change the contents of every row in the table!
- **Not using the correct `WHERE` clause in a `DELETE` statement.** Be careful that you do not leave out the `WHERE` clause and the conditional expression when using a `DELETE` statement. You could delete every row in the table!
- **Not using the correct `WHERE` clause when joining data.** When joining data from multiple tables, be sure to use a `WHERE` clause to specify search criteria that link the appropriate columns. Failure to do so will result in a large set of unrelated data.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This is the technology that makes it possible for a Java application to communicate with a DBMS.
   a. DBMSC
   b. JDBC
   c. JDBMS
   d. JDSQL
2. This is a standard language for working with database management systems.
   a. Java
   b. COBOL
   c. SQL
   d. BASIC
3. The data that is stored in a table is organized in __________.
   a. rows
   b. files
   c. folders
   d. pages
4. The data that is stored in a row is divided into __________.
   a. sections
   b. bytes
   c. columns
   d. tables

5. This is a column that holds a unique value for each row, and can be used to identify specific rows.
   a. ID column
   b. public key
   c. designator column
   d. primary key
6. This type of SQL statement is used to retrieve rows from a table.
   a. `RETRIEVE`
   b. `GET`
   c. `SELECT`
   d. `READ`
7. This contains the results of an SQL `SELECT` statement.
   a. select set
   b. result set
   c. SQL set
   d. collection set
8. This clause allows you to specify search criteria with the `SELECT` statement.
   a. `SEARCH`
   b. `WHERE`
   c. `AS`
   d. `CRITERIA`
9. This is a Java class that is designed to communicate with a specific DBMS.
   a. JDBC driver
   b. DBMS Superclass
   c. DBMS Subclass
   d. Stream converter
10. This is a string listing the protocol that should be used to access a database, the name of the database, and potentially other items.
    a. JDBC driver
    b. JDBC locator
    c. Database URL
    d. Database specifier
11. This method is specified in the `Statement` interface, and should be used to execute a `SELECT` statement.
    a. `execute`
    b. `executeUpdate`
    c. `executeQuery`
    d. `executeSelect`
12. This method is specified in the `Statement` interface, and should be used to execute an `UPDATE` statement.
    a. `execute`
    b. `executeUpdate`
    c. `executeQuery`
    d. `executeSelect`

13. This method is specified in the `Statement` interface, and should be used to execute an `INSERT` statement.
    a. `execute`
    b. `executeUpdate`
    c. `executeQuery`
    d. `executeSelect`
14. This SQL statement is used to insert rows into a table.
    a. `INSERT`
    b. `ADD`
    c. `CREATE`
    d. `UPDATE`
15. This SQL statement is used to remove rows from a table.
    a. `REMOVE`
    b. `ERASE`
    c. `PURGE`
    d. `DELETE`
16. This SQL statement is used to delete an entire table.
    a. `REMOVE`
    b. `DROP`
    c. `PURGE`
    d. `DELETE`
17. This is a column in one table that references a primary key in another table.
    a. secondary key
    b. fake key
    c. foreign key
    d. duplicate key
18. **True/False:** Java comes with its own built-in DBMS.
19. **True/False:** A Java programmer that uses a DBMS to store data does not need to know about the physical structure of the data.
20. **True/False:** You use SQL instead of Java to write entire applications, including the user interface.
21. **True/False:** In SQL, the not-equal-to operator is `!=`, which is the same as in Java.
22. **True/False:** When a `ResultSet` object is initially created, its cursor is pointing at the first row in the result set.
23. **True/False:** In a transaction, it is permissible for only some of the database updates to be made.
24. **True/False:** The term *rollback* refers to undoing changes to a database.

## Find the Error

1. Find the error in the following SQL statement.

```
SELECT * FROM Coffee WHERE Description = "French Roast Dark"
```

2. Find the error in the following SQL statement.

```
SELECT * FROM Coffee WHERE ProdNum != '14-001'
```

3. Find the error in the following Java code. Assume that `conn` references a valid `Connection` object.

```
// This code has an error!!!
String sql = "SELECT * FROM Coffee";
Statement stmt = conn.createStatement();
ResultSet result = stmt.execute(sql);
```

## Algorithm Workbench

1. What SQL data types correspond with the following Java types?
    - `int`
    - `float`
    - `String`
    - `double`
2. Look at the following SQL statement.

    ```
    SELECT Name FROM Employee
    ```

    What is the name of the table from which this statement is retrieving data?
    What is the name of the column that is being retrieved?

*For questions 3 through 12, assume that a database has a table named* `Stock`*, with the following columns:*

| Column Name | Type |
|---|---|
| `TradingSymbol` | `CHAR(10)` |
| `CompanyName` | `CHAR(25)` |
| `NumShares` | `INT` |
| `PurchasePrice` | `DOUBLE` |
| `SellingPrice` | `DOUBLE` |

3. Write a `SELECT` statement that will return all of the columns from every row in table.
4. Write a `SELECT` statement that will return the `TradingSymbol` column from every row in table.
5. Write a `SELECT` statement that will return the `TradingSymbol` column and the `NumShares` column from every row in table.
6. Write a `SELECT` statement that will return the `TradingSymbol` column only from the rows where `PurchasePrice` is greater than 25.00.
7. Write a `SELECT` statement that will return all of the columns from the rows where `TradingSymbol` starts with “SU”.
8. Write a `SELECT` statement that will return the `TradingSymbol` column only from the rows where `SellingPrice` is greater than `PurchasePrice`, and `NumShares` is greater than 100.
9. Write a `SELECT` statement that will return the `TradingSymbol` column and the `NumShares` column only from the rows where `SellingPrice` is greater than `PurchasePrice`, and `NumShares` is greater than 100. The results should be sorted by the `NumShares` column, in ascending order.

10. Write an SQL statement that will insert a new row into the `Stock` table. The row should have the following column values:

    `TradingSymbol`: XYZ
    `CompanyName`: “XYZ Company”
    `NumShares`: 150
    `PurchasePrice`: 12.55
    `SellingPrice`: 22.47

11. Write an SQL statement that does the following: For each row in the `Stock` table, if the `TradingSymbol` column is “XYZ”, change it to “ABC”.

12. Write an SQL statement that will delete rows in the `Stock` table where the number of shares is less than 10.

13. Assume that the following declaration exists.

    ```
    final String DB_URL = "jdbc:derby:CoffeeDB";
    ```

    The string referenced by `DB_URL` is a database URL. Write a statement that uses this string to get a connection to the database.

14. Assuming that `conn` references a valid `Connection` object, write code to create a `Statement` object. (Do not be concerned about result set scrolling or concurrency.)

15. Look at the following declaration.

    ```
    String sql = "SELECT * FROM Coffee WHERE Price . 10.00";
    ```

    Assume also that `stmt` references a valid `Statement` object. Write code that executes the SQL statement referenced by the `sql` variable.

16. Assume that the following code is used to retrieve data from the `CoffeeDB` database’s `Coffee` table. Write the code that should appear inside the loop to display the contents of the result set.

    ```
    String sql = "SELECT * FROM Coffee";
    Connection conn = DriverManager.getConnection(DB_URL);
    Statement stmt = conn.createStatement();
    ResultSet result = stmt.executeQuery(sql);
    while (result.next())
    {
       // Finish this code!!
    }
    stmt.close();
    conn.close();
    ```

17. Write an SQL statement to create a table named `Car`. The `Car` table should have the columns to hold a car’s manufacturer, year model, and a 20-character vehicle ID number.

18. Write an SQL statement to delete the `Car` table you created in Algorithm Workbench 17.

## Short Answer

1. If you are writing an application to store the customer and inventory records for a large business, why would you not want to use traditional text or binary files?

2. You hear a fellow classmate say the following: "JDBC is a standard language for working with database management systems. It was invented at IBM." Are these statements correct, or is he confusing JDBC with something else?
3. When we speak of database organization, we speak of such things as rows, tables, and columns. Describe how the data in a database is organized into these conceptual units.
4. What is a primary key?
5. What is a result set?
6. What are the relational operators in SQL for the following comparisons?

   Greater-than
   Less-than
   Greater-than or equal-to
   Less-than or equal-to
   Equal-to
   Not equal-to
7. What is the number of the first row in a table? What is the number of the first column in a table?
8. What is metadata? What is result set metadata? When is result set metadata useful?
9. What is a foreign key?

## Programming Challenges

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

VideoNote
The Customer Inserter Problem

### 1. Customer Inserter

Write an application that connects to the `CoffeeDB` database, and allows the user to insert a new row into the `Customer` table.

### 2. Customer Updater

Write an application that connects to the `CoffeeDB` database, and allows the user to select a customer, then change any of that customer's information. (You should not attempt to change the customer number, because it is referenced by the `UnpaidOrder` table.)

### 3. Unpaid Order Sum

Write an application that connects to the `CoffeeDB` database, then calculates and displays the total amount owed in unpaid orders. This will be the sum of each row's `Cost` column.

### 4. Population Database

Make sure you have downloaded the book's source code from the companion Web site at **`www.pearson.com/gaddis`**. In this chapter's source code folder you will find a program named `CreateCityDB.java`. Compile and run the program. The program will create a Java

DB or Apache Derby database named `CityDB`. The `CityDB` database will have a table named `City`, with the following columns:

| Column Name | Data Type |
|---|---|
| `CityName` | `CHAR (50)` |
| *Primary key* | |
| `Population` | `DOUBLE` |

The `CityName` column stores the name of a city and the `Population` column stores the population of that city. After you run the `CreateCityDB.java` program, the `City` table will contain 20 rows with various cities and their populations.

Next, write a program that connects to the `CityDB` database, and allows the user to select any of the following operations:

- Sort the list of cities by population, in ascending order.
- Sort the list of cities by population, in descending order.
- Sort the list of cities by name.
- Get the total population of all the cities.
- Get the average population of all the cities.
- Get the highest population.
- Get the lowest population.

### 5. `Personnel` Database Creator

Write an application that creates a database named `Personnel`. The database should have a table named `Employee`, with columns for employee ID, name, position, and hourly pay rate. The employee ID should be the primary key. Insert at least five sample rows of data into the `Employee` table.

### 6. Employee Inserter

Write a GUI application that allows the user to add new employees to the `Personnel` database you created in Programming Challenge 5.

### 7. Employee Updater

Write a GUI application that allows the user to look up an employee in the `Personnel` database you created in Programming Challenge 5. The user should be able to change any of the employee's information except employee ID, which is the primary key.

### 8. `PhoneBook` Database

Write an application that creates a database named `PhoneBook`. The database should have a table named `Entries`, with columns for a person's name and phone number.

Next, write an application that lets the user add rows to the `Entries` table, look up a person's phone number, change a person's phone number, and delete specified rows.

CHAPTER

# 23

# A First Look at GUI Applications with Swing

**TOPICS**

23.1 Introduction
23.2 Creating Windows
23.3 Layout Managers
23.4 Radio Buttons and Check Boxes
23.5 Borders
23.6 Focus on Problem Solving: Extending Classes from `JPanel`
23.7 Splash Screens
23.8 Using Console Output to Debug a GUI Application
23.9 Common Errors to Avoid

> **NOTE:** This chapter discusses GUI development using the Swing classes. Oracle has announced that JavaFX is replacing Swing as the standard GUI library for Java. Swing will remain part of the Java API for the foreseeable future, however, so we are providing this chapter for you to use as you make the transition from Swing to JavaFX. To learn about JavaFX, see Chapters 12, 13, and 14.

## 23.1 Introduction

**CONCEPT: In Java, you use the Java Foundation Classes (JFC) to create a graphical user interface for your application. Within the JFC you use the Abstract Windowing Toolkit (AWT) or Swing classes to create a graphical user interface.**

In this chapter, we discuss the basics of creating a Java application with a *graphical user interface* or *GUI* (pronounced "gooey"). A GUI is a graphical window or a system of graphical windows that is presented by an application for interaction with the user. In addition to accepting input from the keyboard, GUIs typically accept input from a mouse as well.

A window in a GUI commonly consists of several *components* that present data to the user and/or allow interaction with the application. Some of the common GUI components are buttons, labels, text fields, check boxes, and radio buttons. Figure 23-1 shows an example of a window with a variety of components. Table 23-1 describes the components that appear in the window.

**Figure 23-1** Various GUI components (Oracle Corporate Counsel)

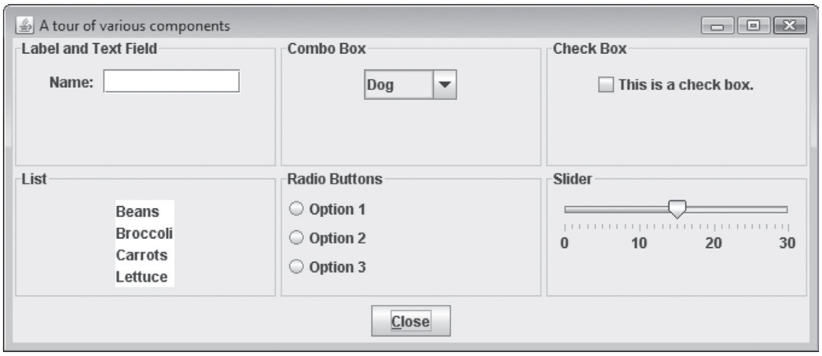

A tour of various components
Label and Text Field
Name:
Combo Box
Dog
Check Box
☐ This is a check box.
List
Beans
Broccoli
Carrots
Lettuce
Radio Buttons
○ Option 1
○ Option 2
○ Option 3
Slider
0
10
20
30
Close

**Table 23-1** Some GUI components

| Component | Description |
|---|---|
| Label | An area that can display text. |
| Text field | An area in which the user may type a single line of input from the keyboard. |
| Combo box | A component that displays a drop-down list of items from which the user may select. A combo box also provides a text field in which the user may type input. It is called a combo box because it is the combination of a list and a text field. |
| Check box | A component that has a box that may be checked or unchecked. |
| List | A list from which the user may select an item. |
| Radio button | A component that can be either selected or deselected. Radio buttons usually appear in groups and allow the user to select one of several options. |
| Slider | A component that allows the user to select a value by moving a slider along a track. |
| Button | A button that can cause an action to occur when it is clicked. |

## The JFC, AWT, and Swing

Java programmers use the *Java Foundation Classes (JFC)* to create GUI applications. The JFC consists of several sets of classes, many of which are beyond the scope of this book. The two sets of JFC classes that we focus on are the AWT and Swing classes. First, we discuss the differences between them.

Java has been equipped, since its earliest version, with a set of classes for drawing graphics and creating GUIs. These classes are part of the *Abstract Windowing Toolkit (AWT)*. The AWT allows programmers to create applications and applets that interact with the user via windows and other GUI components.

Programmers are limited in what they can do with the AWT classes, however. This is because the AWT classes do not actually draw user interface components on the screen. Instead, the AWT classes communicate with another layer of software, known as the *peer classes*, which directs the underlying operating system to draw its own built-in components. Each version of Java that is developed for a particular operating system has its own set of peer classes. Although this means that Java programs have a look that is consistent with other applications on the same system, it also leads to some problems.

One problem is that not all operating systems offer the same set of GUI components. For example, one operating system might provide a sophisticated slider bar component that is not found on any other platform. Other operating systems might have their own unique components as well. In order for the AWT to retain its portability, it has to offer only those components that are common to all the operating systems that support Java.

Another problem is in the behavior of components across various operating systems. A component on one operating system might have slightly different behavior than the same component on a different operating system. In addition, the peer classes for some operating systems reportedly have bugs. As a result, programmers cannot be completely sure how their AWT programs will behave on different operating systems until they test each one.

A third problem is that programmers cannot easily customize the AWT components. Because these components rely on the appearance and behavior of the underlying operating system components, there is little that can be done by the programmer to change their properties.

To remedy these problems, Swing was introduced with the release of Java 2. *Swing* is a library of classes that do not replace the AWT, but provide an improved alternative for creating GUI applications and applets. Very few of the Swing classes rely on an underlying system of peer classes. Instead, Swing draws most of its own components on the screen. This means that Swing components can have a consistent look and predictable behavior on any operating system.

**NOTE:** Swing applications can have the look of a specific operating system. The programmer may choose from a variety of “look and feel” themes.

Swing components can also be easily customized. The Swing library provides many sophisticated components that are not found in the AWT. In this chapter and in Chapter 24, we primarily use Swing to develop GUI applications. In Chapter 25, we use AWT to develop applets.

**NOTE:** AWT components are commonly called heavyweight components because they are coupled with their underlying peer classes. Very few of the Swing components are coupled with peer classes, so they are referred to as lightweight components.

### Event-Driven Programming

Programs that operate in a GUI environment must be *event-driven*. An *event* is an action that takes place within a program, such as the clicking of a button. Part of writing a GUI application is creating event listeners. An *event listener* is an object that automatically executes one of its methods when a specific event occurs. If you wish for an application to perform an operation when a particular event occurs, you must create an event listener object that responds when that event takes place.

### The `javax.swing` and `java.awt` Packages

In this chapter, we use the Swing classes for all of the graphical components that we create in our GUIs. The Swing classes are part of the `javax.swing` package. (Take note of the letter `x` that appears after the word `java`.) The following `import` statement will be used in every applicaton:

```
import javax.swing.*;
```

We also use some of the AWT classes to determine when events, such as the clicking of a mouse, take place in our applications. The AWT classes are part of the `java.awt` package. (Note that there is no `x` after `java` in this package name.) Programs that use the AWT classes will have the following `import` statement:

```
import java.awt.*;
```

## 23.2 Creating Windows

**CONCEPT: You can use Swing classes to create windows containing various GUI components.**

VideoNote
Creating a Simple GUI Application

The `JOptionPane` dialog boxes that you learned about in Chapter 2 allow you to easily display messages and gather input. If an application is to provide a full graphical user interface, however, much more is needed. Often, applications need one or more windows with various components that allow the user to enter and/or select data and interact with the application. For example, the window that is displayed in Figure 23-1 has several different components within it.

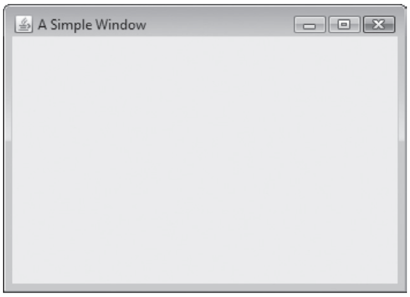
A window is a component, but because a window contains other components, it is more appropriately considered a container. A *container* is simply a component that holds other components. In GUI terminology, a container that can be displayed as a window is known as a *frame*. A frame appears as a basic window that has a border around it, a title bar, and a set of buttons for minimizing, maximizing, and closing the window. In a Swing application, you create a frame object from the `JFrame` class.

There are a number of steps involved in creating a window, so let's look at an example. The program in Code Listing 23-1 displays the window shown in Figure 23-2.

**Code Listing 23-1** (`ShowWindow.java`)

```java
import javax.swing.*;    // Needed for Swing classes

/**
   This program displays a simple window with a title. The
   application exits when the user clicks the close button.
*/

public class ShowWindow
{
   public static void main(String[] args)
   {
      final int WINDOW_WIDTH = 350;    // Window width in pixels
      final int WINDOW_HEIGHT = 250;   // Window height in pixels

      // Create a window.
      JFrame window = new JFrame();

      // Set the title.
      window.setTitle("A Simple Window");

      // Set the size of the window.
      window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Display the window.
      window.setVisible(true);
   }
}
```

**Figure 23-2** Window displayed by `ShowWindow.java` (Oracle Corporate Counsel)



The window shown in Figure 23-2 was produced on a system running Microsoft Windows. Notice that the window has a border and a title bar with "A Simple Window" displayed in it. In addition, it has the standard Microsoft Windows buttons in the upper-right corner: a minimize button, a maximize button, and a close button. These standard features are sometimes referred to as *decorations*. If you run this program, you will see the window displayed on your screen. When you click on the close button, the window disappears and the program terminates.

Let's take a closer look at the code. First, notice that the following `import` statement is used in line 1:

```
import javax.swing.*;   // Needed for Swing classes
```

Any program that uses a Swing class, such as `JFrame`, must have this `import` statement. In lines 12 and 13 the two constants `WINDOW_WIDTH` and `WINDOW_HEIGHT` are declared as follows:

```
final int WINDOW_WIDTH = 350;   // Window width in pixels
final int WINDOW_HEIGHT = 250;  // Window height in pixels
```

We use these constants later in the program to set the size of the window. The window's size is measured in pixels. A *pixel* is one of the small dots that make up a screen display; the resolution of your monitor is measured in pixels. For example, if your monitor's resolution is 1024 by 768, that means the width of your screen is 1024 pixels, and the height of your screen is 768 pixels.

Next, we create an instance of the `JFrame` class with the following statement in line 16:

```
JFrame window = new JFrame();
```

This statement creates a `JFrame` object in memory and assigns its address to the `window` variable. This statement does not display the window on the screen, however. A `JFrame` is initially invisible.

In line 19 we call the `JFrame` object's `setTitle` method as follows:

```
window.setTitle("A Simple Window");
```

The string that is passed as an argument to `setTitle` will appear in the window's title bar when it is displayed. In line 22 we call the `JFrame` object's `setSize` method to set the window's size as follows:

```
window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
```

The two arguments passed to `setSize` specify the window's width and height in pixels. In this program we pass the constants `WINDOW_WIDTH` and `WINDOW_HEIGHT`, which we declared earlier, to set the size of the window to 350 pixels by 250 pixels.

In line 25 we specify the action that we wish to take place when the user clicks on the close button, which appears in the upper-right corner of the window as follows:

```
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

There are a number of actions that can take place when the user clicks on the close button. The `setDefaultCloseOperation` method takes an `int` argument, which specifies the action. In this statement, we pass the constant `JFrame.EXIT_ON_CLOSE`, which causes the application to end with a `System.exit` method call. If we had passed `JFrame.HIDE_ON_CLOSE`, the window would be hidden from view, but the application would not end. The default action is `JFrame.HIDE_ON_CLOSE`.

Last, in line 28, we use the following code to display the window:

```
window.setVisible(true);
```

The `setVisible` method takes a `boolean` argument. If the argument is `true`, the window is made visible. If the argument is `false`, the window is hidden.

## Using Inheritance to Extend the `JFrame` Class

The program in Code Listing 23-1 performs a very simple operation: It creates an instance of the `JFrame` class and displays it. Most of the time, your GUI applications will be much more involved than this. As you progress through this chapter, you will add numerous components and capabilities to the windows that you create.

Instead of simply creating an instance of the `JFrame` class, as shown in Code Listing 23-1, a more common technique is to use inheritance to create a new class that extends the `JFrame` class.

**IF YOU'VE SKIPPED AHEAD TO THIS CHAPTER:** This chapter is written so that you can skip ahead to it any time after Chapter 6. Reading about inheritance and interfaces in Chapter 10 would be helpful; but if you have not read that material yet, the following summarizes what you need to know for this chapter.

When a new class *extends* an existing class, it inherits many of the existing class's members just as if they were part of the new class. For example, you saw how the program in Code Listing 23-1 created a `JFrame` object and then called four of its methods: `setTitle`, `setSize`, `setDefaultCloseOperation`, and `setVisible`. These methods are all members of the `JFrame` class. If you create a new class that extends the `JFrame` class, the new class will automatically inherit these methods. Then these methods can be called from an instance of the new class just as if they were written into its declaration. You can add your own custom code to the new class, making it a specialized, or extended, version of the `JFrame` class. Programs can then create instances of your new specialized class instead of the more generic `JFrame` class.

Let's look at the `SimpleWindow` class in Code Listing 23-2. This is an example of a class that extends the `JFrame` class.

**Code Listing 23-2** (`SimpleWindow.java`)

```
import javax.swing.*;    // Needed for Swing classes

/**
   This class extends the JFrame class. Its constructor displays
   a simple window with a title. The application exits when the
```

```
 6     user clicks the close button.
 7  */
 8
 9  public class SimpleWindow extends JFrame
10  {
11     /**
12        Constructor
13     */
14
15     public SimpleWindow()
16     {
17        final int WINDOW_WIDTH = 350;   // Window width in pixels
18        final int WINDOW_HEIGHT = 250;  // Window height in pixels
19
20        // Set this window's title.
21        setTitle("A Simple Window");
22
23        // Set the size of this window.
24        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
25
26        // Specify what happens when the close button is clicked.
27        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28
29        // Display the window.
30        setVisible(true);
31     }
32  }
```

Notice the class header in line 9 as follows:

```
public class SimpleWindow extends JFrame
```

The words `extends JFrame` indicate that the `SimpleWindow` class extends the `JFrame` class. This means that the `SimpleWindow` class inherits members of the `JFrame` class, such as the `setTitle`, `setSize`, `setDefaultCloseOperation`, and `setVisible` methods, just as if they were written into the `SimpleWindow` class declaration. Now look at the constructor. In lines 17 and 18 we declare the `WINDOW_WIDTH` and `WINDOW_HEIGHT` constants, which will be used to establish the size of the window as follows:

```
final int WINDOW_WIDTH = 350;   // Window width in pixels
final int WINDOW_HEIGHT = 250;  // Window height in pixels
```

In line 21 we call the `setTitle` method to set the text for the window's title bar as follows:

```
setTitle("A Simple Window");
```

Notice that we are calling the method without an object reference and a dot preceding it. This is because the method was inherited from the `JFrame` class, and we can call it just as if it were written into the `SimpleWindow` class declaration.

The rest of the constructor calls the `setSize`, `setDefaultCloseOperation`, and `setVisible` methods. All that is necessary to display the window is to create an instance of the `SimpleWindow` class, as shown in the program in Code Listing 23-3. When this program runs, the window that was previously shown in Figure 23-2 is displayed. Remember, the `SimpleWindow` class is an extended version of the `JFrame` class. When we create an instance of the `SimpleWindow` class, we are really creating an instance of the `JFrame` class, with some customized code added to its constructor.

**Code Listing 23-3** (`SimpleWindowDemo.java`)

```
/**
   This program creates an instance of the
   SimpleWindow class.
*/

public class SimpleWindowDemo
{
   public static void main(String[] args)
   {
      SimpleWindow myWindow = new SimpleWindow();
   }
}
```

## Equipping GUI Classes with a `main` Method

You know that a Java application always starts execution with a static method named `main`. The previous example consists of two separate files:

- `SimpleWindow.java`: This file contains the `SimpleWindow` class, which defines a GUI window.
- `SimpleWindowDemo.java`: This file contains a static `main` method that creates an object of the GUI window class, thus displaying it.

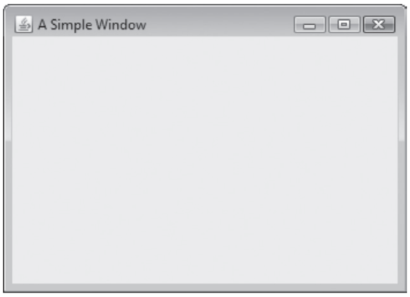
The purpose of the `SimpleWindowDemo.java` file is simply to create an instance of the `SimpleWindow` class. It is possible to eliminate the second file, `SimpleWindowDemo.java`, by writing the static `main` method directly into the `SimpleWindow.java` file. The `EmbeddedMain` class in Code Listing 23-4 shows an example.

**Code Listing 23-4** (`EmbeddedMain.java`)

```
import javax.swing.*; // Needed for Swing classes

/**
   This class defines a GUI window and has its own
   main method.
*/

public class EmbeddedMain extends JFrame
{
   final int WINDOW_WIDTH = 350;    // Window width in pixels
   final int WINDOW_HEIGHT = 250;   // Window height in pixels
12
13    /**
14       Constructor
15    */
16
17    public EmbeddedMain()
18    {
19       // Set this window's title.
20       setTitle("A Simple Window");
21
22       // Set the size of this window.
23       setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
24
25       // Specify what happens when the close button is clicked.
26       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27
28       // Display the window.
29       setVisible(true);
30    }
31
32    /**
33       The main method creates an instance of the EmbeddedMain
34       class, which causes it to display its window.
35    */
36
37    public static void main(String[] args)
38    {
39       EmbeddedMain em = new EmbeddedMain();
40    }
41 }
```

The `EmbeddedMain` class contains its own static `main` method (in lines 37 through 40), which creates an instance of the class. Notice that the `main` method has exactly the same header as any other static `main` method that we have written. We can compile the *EmbeddedMain.java* file and then run the resulting *.class* file. When we do, we see the window shown in Figure 23-3.

**Figure 23-3** Window displayed by the `EmbeddedMain` class (Oracle Corporate Counsel)



Notice that in line 39 the `main` method declares a variable named `em` to reference the instance of the class. Once the instance is created, however, the variable is not used again. Because we do not need the variable, we can instantiate the class *anonymously* as shown here:

```
public static void main(String[] args)
{
   new EmbeddedMain();
}
```

In this version of the method, an instance of the `EmbeddedMain` class is created in memory, but its address is not assigned to any reference variable.

## Adding Components to a Window

Swing provides numerous GUI components that can be added to a window. Three fundamental components are the label, the text field, and the button. These are summarized in Table 23-2.

**Table 23-2** Label, text field, and button controls

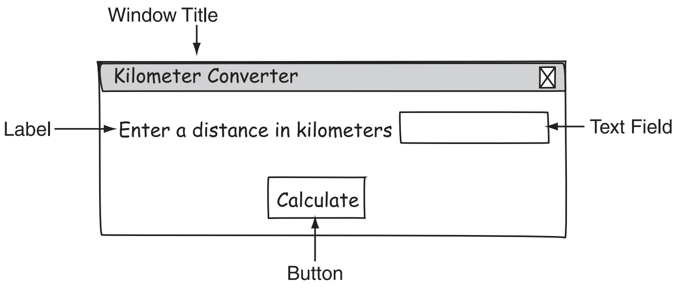
| Component | Swing Class | Description |
|---|---|---|
| Label | `JLabel` | An area that can display text |
| Text field | `JTextField` | An area in which the user may type a single line of input from the keyboard |
| Button | `JButton` | A button that can cause an action to occur when it is clicked |

In Swing, labels are created with the `JLabel` class, text fields are created with the `JTextField` class, and buttons are created with the `JButton` class. To demonstrate these components, we will build a simple GUI application: The Kilometer Converter. This application will present a window in which the user will be able to enter a distance in kilometers, and then click a button to see that distance converted to miles. The conversion formula is as follows:

$$Miles = Kilometers \times 0.6214$$

When designing a GUI application, it is usually helpful to draw a sketch showing the window you are creating. Figure 23-4 shows a sketch of what the Kilometer Converter application's window will look like. As you can see from the sketch, the window will have a label, a text field, and a button. When the user clicks the button, the distance in miles will be displayed in a separate `JOptionPane` dialog box.

**Figure 23-4** Sketch of the Kilometer Converter window (Oracle Corporate Counsel)

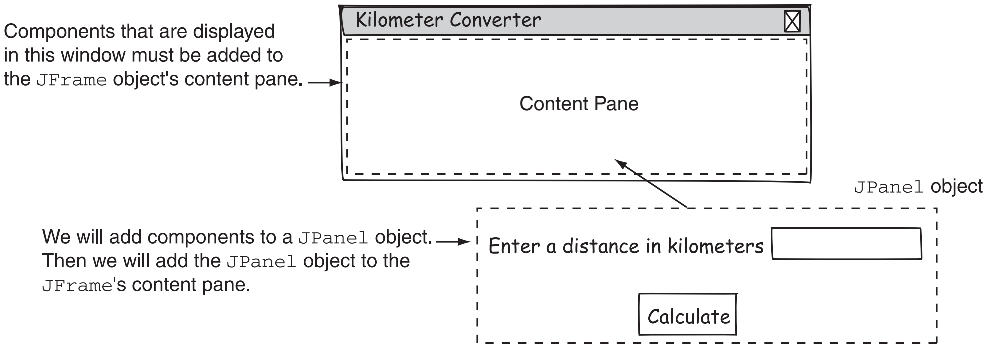

Window Title
Kilometer Converter
Label
Enter a distance in kilometers
Text Field
Calculate
Button

### Content Panes and Panels

Before we start writing code, you should be familiar with content panes and panels. A *content pane* is a container that is part of every `JFrame` object. You cannot see the content pane and it does not have a border, but any component that is to be displayed in a `JFrame` must be added to its content pane.

A *panel* is also a container that can hold GUI components. Unlike `JFrame` objects, panels cannot be displayed by themselves; however, they are commonly used to hold and organize collections of related components. With Swing, you create panels with the `JPanel` class. In our Kilometer Converter application, we will create a panel to hold the label, text field, and button. Then we will add the panel to the `JFrame` object's content pane. This is illustrated in Figure 23-5.

**Figure 23-5** A panel is added to the content pane (Oracle Corporate Counsel)


Components that are displayed in this window must be added to the JFrame object's content pane.
Kilometer Converter
Content Pane
JPanel object
We will add components to a JPanel object. Then we will add the JPanel object to the JFrame's content pane.
Enter a distance in kilometers
Calculate

Code Listing 23-5 shows the initial code for the `KiloConverter` class. We will be adding to this code as we develop the application. This version of the class is stored in the source code folder *Chapter 23\KiloConverter Phase 1*.

**Code Listing 23-5** (`KiloConverter.java`)

```
import javax.swing.*;

/**
   The KiloConverter class displays a JFrame that
   lets the user enter a distance in kilometers. When
   the Calculate button is clicked, a dialog box is
   displayed with the distance converted to miles.
*/

public class KiloConverter extends JFrame
{
   private JPanel panel;                 // To reference a panel
   private JLabel messageLabel;          // To reference a label
   private JTextField kiloTextField;     // To reference a text field
   private JButton calcButton;           // To reference a button
   private final int WINDOW_WIDTH = 310; // Window width
   private final int WINDOW_HEIGHT = 100;// Window height

   /**
      Constructor
   */

   public KiloConverter()
   {
      // Set the window title.
      setTitle("Kilometer Converter");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Build the panel and add it to the frame.
      buildPanel();

      // Add the panel to the frame's content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      The buildPanel method adds a label, a text field,
      and a button to a panel.
   */

   private void buildPanel()
   {
      // Create a label to display instructions.
      messageLabel = new JLabel("Enter a distance " +
                                "in kilometers");
54
55       // Create a text field 10 characters wide.
56       kiloTextField = new JTextField(10);
57
58       // Create a button with the caption "Calculate".
59       calcButton = new JButton("Calculate");
60
61       // Create a JPanel object and let the panel
62       // field reference it.
63       panel = new JPanel();
64
65       // Add the label, text field, and button
66       // components to the panel.
67       panel.add(messageLabel);
68       panel.add(kiloTextField);
69       panel.add(calcButton);
70    }
71
72    /**
73       main method
74    */
75
76    public static void main(String[] args)
77    {
78       new KiloConverter();
79    }
80 }
```

Let's take a closer look at this class. First, notice in line 10 that the `KiloConverter` class extends the `JFrame` class as follows:

```
public class KiloConverter extends JFrame
```

Next, in lines 12 through 17, notice in the following that the class declares a number of fields, and according to good class design principles, the fields are private:

```
private JPanel panel;                  // To reference a panel
private JLabel messageLabel;           // To reference a label
private JTextField kiloTextField;      // To reference a text field
private JButton calcButton;            // To reference a button
private final int WINDOW_WIDTH = 310;  // Window width
private final int WINDOW_HEIGHT = 100; // Window height
```

The statement in line 12 declares a `JPanel` reference variable named `panel`, which we will use to reference the panel that will hold the other components. The `messageLabel` variable, declared in line 13, will reference a `JLabel` object that displays a message instructing the user to enter a distance in kilometers. The `kiloTextField` variable, declared in line 14, will reference a `JTextField` object that will hold a value typed by the user. The `calcButton` variable, declared in line 15, will reference a `JButton` object that will calculate and display the kilometers converted to miles when clicked. The `WINDOW_WIDTH` and `WINDOW_HEIGHT` fields, declared in lines 16 and 17, are constants that hold the width and height of the window.

Now let's look at the constructor. In line 26 the `setTitle` method, which was inherited from the `JFrame` class, is called to set the text for the window's title bar. Next, in line 29, the inherited `setSize` method is called to establish the size of the window. In line 32, the inherited `setDefaultCloseOperation` method is called to establish the action that should occur when the window's close button is clicked.

Line 35 calls the `buildPanel` method. The `buildPanel` method is defined in this class, in lines 49 through 70. The purpose of the `buildPanel` method is to create a label, a text field, and a button, and then add those components to a panel. Let's look at the method.

First, look at the method header in line 49 and notice that it is declared private. When a method is private, only other methods in the same class can call it. This method is not meant to be called by code outside the class, so it is declared private. In lines 52 and 53, the method uses the following statement to create a `JLabel` object and assign its address to the `message` field:

```
messageLabel = new JLabel("Enter a distance " +
                          "in kilometers");
```

The string that is passed to the `JLabel` constructor is the text that will be displayed in the label. The following statement appears in line 56. It creates a `JTextField` object, and assigns its address to the `kiloTextField` field:

```
kiloTextField = new JTextField(10);
```

The argument that is passed to the `JTextField` constructor is the width of the text field in columns. One column is enough space to hold the letter "m," which is the widest letter in the alphabet.

The following statement appears in line 59; it creates a `JButton` object, and assigns its address to the `calcButton` field:

```
calcButton = new JButton("Calculate");
```

The string that is passed as an argument to the `JButton` constructor is the text that will be displayed on the button.

Next, in line 63, the method uses the following statement to create a `JPanel` object and assign its address to the `panel` field, which is a private field in the class:

```
panel = new JPanel();
```

A `JPanel` object is used to hold other components. You add a component to a `JPanel` object with the `add` method. The following code, in lines 67 through 69, adds the objects referenced by the `messageLabel`, `kiloTextField`, and `calcButton` variables to the `JPanel` object:

```
panel.add(messageLabel);
panel.add(kiloTextField);
panel.add(calcButton);
```

At this point, the panel is fully constructed in memory. The `buildPanel` method ends, and control returns to the class constructor. Here's the next statement in the constructor, which appears in line 38:

```
add(panel);
```

This statement calls the `add` method, which was inherited from the `JFrame` class. The purpose of the `add` method is to add an object to the content pane. This statement adds the object referenced by `panel` to the content pane.

The constructor's last statement, in line 41, calls the inherited `setVisible` method to display the window on the screen as follows:

```
setVisible(true);
```

The class has a static `main` method, which appears in lines 76 through 79. Line 78 creates an instance of the `KiloConverter` class. When this program is executed, the window shown in Figure 23-6 is displayed on the screen.

**Figure 23-6** Kilometer Converter window (Oracle Corporate Counsel)

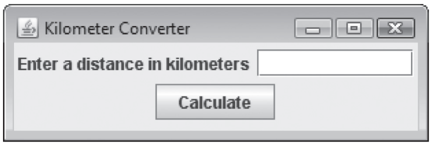
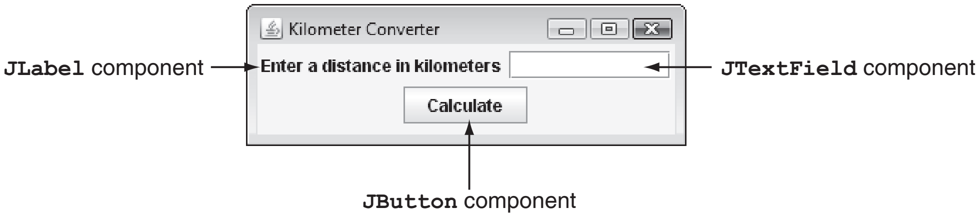


Figure 23-7 shows the window again, this time pointing out each of the components.

Although you can type input into the text field, the application does nothing when you click the Calculate button because we have not written an event handler that will execute when the button is clicked. That's the next step.

**Figure 23-7** Components in the Kilometer Converter window (Oracle Corporate Counsel)



**NOTE:** Recall that the size of the window in the `KiloConverter` class is set to 310 pixels wide by 100 pixels high. This is set with the `WINDOW_WIDTH` and `WINDOW_HEIGHT` constants. Figures 23-6 and 23-7 show the window as it appears on a system set at a video resolution of 1024 by 768 pixels. If your video resolution is lower, the window might not appear exactly as shown in the figures. If this is the case, you can increase the values of the `WINDOW_WIDTH` and `WINDOW_HEIGHT` constants and recompile the program. This is true for other applications in this chapter as well.

## Handling Events with Action Listeners

VideoNote
Handling Events

An *event* is an action that takes place within a program, such as the clicking of a button. When an event takes place, the component that is responsible for the event creates an event object in memory. The *event object* contains information about the event. The component that generated the event object is known as the *event source*. For example, when the user clicks a button, the `JButton` component generates an event object. The `JButton` component that generated the event object is the event source.

But what happens to the event object once it is generated by a source component? It is possible that the source component is connected to one or more event listeners. An *event listener* is an object that responds to events. If the source component is connected to an event listener, then the event object is automatically passed, as an argument, to a specific method in the event listener. The method then performs any actions that it was programmed to perform in response to the event. This process is sometimes referred to as *event firing*.

When you are writing a GUI application, it is your responsibility to write the classes for the event listeners that your application needs. For example, if you write an application with a `JButton` component, an event will be generated each time the user clicks the button. Therefore, you should write an event listener class that can handle the event. In your application you would create an instance of the event listener class and connect it to the `JButton` component. Before looking at a specific example, we must discuss two important topics that arise when writing event listeners: private inner classes and interfaces.

### Writing Event Listener Classes as Private Inner Classes

Java allows you to write a class definition inside of another class definition. A class that is defined inside of another class is known as an *inner class*. Figure 23-8 illustrates a class definition inside of another class definition.

**Figure 23-8** A class with an inner class

```
public class Outer
{
   Fields and methods of the Outer
   class appear here.

   private class Inner
   {
      Fields and methods of the Inner
      class appear here.
   }
}
```

When an inner class is private, as shown in the figure, it is accessible only to code in the class that contains it. For example, the `Inner` class shown in the figure would be accessible only to methods that belong to the `Outer` class. Code outside the `Outer` class would not be able to access the `Inner` class. A common technique for writing an event listener class is to write it as a private inner class, inside the class that creates the GUI. Although this is not the only way to write event listener classes, it is the approach we take in this book.

### Event Listeners Must Implement an Interface

There is a special requirement that all event listener classes must meet: They must *implement an interface*.

We discussed interfaces in detail in Chapter 10, but in case you haven't read that material, you can think of an interface as something like a class, containing one or more method headers. Interfaces do not have actual methods, however, only their headers. When you write a class that implements an interface, you are agreeing that the class will have all of the methods that are specified in the interface.

Java provides numerous interfaces that you can use with event listener classes. There are several different types of events that can occur within a GUI application, and the specific interface that you use depends on the type of event you want to handle. `JButton` components generate *action events*, and an event listener class that can handle action events is also known as an *action listener* class. When you write an action listener class for a `JButton` component, it must implement an interface known as `ActionListener`. In case you are curious, this is what the code for the `ActionListener` interface looks like:

```
public interface ActionListener
{
   public void actionPerformed(ActionEvent e);
}
```

As you can see, the `ActionListener` interface contains the header for only one method: `actionPerformed`. Notice that the method has `public` access, is `void`, and has a parameter of the `ActionEvent` type. When you write a class that implements this interface, it *must* have a method named `actionPerformed`, with a header exactly like the one in the interface.

**NOTE:** The *ActionListener* interface, as well as other event listener interfaces, is in the *java.awt.event* package. We will use the following *import* statement in order to use those interfaces:

```
import java.awt.event.*;
```

You use the `implements` key word in a class header to indicate that it implements an interface. Here is an example of a class named `MyButtonListener` that implements the `ActionListener` interface:

```
private class MyButtonListener implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      Write code here to handle the event.
   }
}
```

Remember, when you write a class that implements an interface, you are "promising" that the class will have the methods specified in the interface. Notice that this class lives up to its promise. It has a method named `actionPerformed`, with a header that matches the `actionPerformed` header in the `ActionListener` interface exactly.

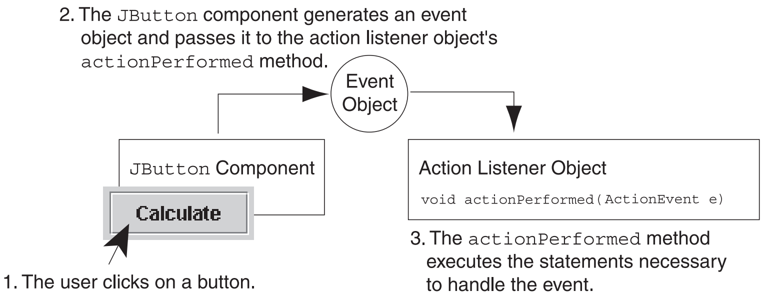
**NOTE:** In your action listener class, the only part of the `actionPerformed` method header that does not have to match that which is shown in the `ActionListener` interface exactly is the name of the parameter variable. Instead of using the name `e`, you can use any legal variable name that you wish.

### Registering an Event Listener Object

Once you have written an event listener class, you can create an object of that class, and then connect the object with a GUI component. The process of connecting an event listener object to a GUI component is known as *registering* the event listener.

When a `JButton` component generates an event, it automatically executes the `actionPerformed` method of the event listener object that is registered with it, passing the event object as an argument. This is illustrated in Figure 23-9.

**Figure 23-9** A `JButton` component firing an action event (Oracle Corporate Counsel)


2. The JButton component generates an event object and passes it to the action listener object's actionPerformed method.
Event Object
JButton Component
Calculate
Action Listener Object
void actionPerformed(ActionEvent e)
1. The user clicks on a button.
3. The actionPerformed method executes the statements necessary to handle the event.

## Writing an Event Listener for the `KiloConverter` Class

Now that we've gone over the basics of event listeners, let's continue to develop the `KiloConverter` class. Code Listing 23-6 shows the class with an action listener added to it. This version of the class is stored in the source code folder *Chapter 23\KiloConverter Phase 2*. The action listener is a private inner class named `CalcButtonListener`. The new code is shown in bold.

**Code Listing 23-6** (`KiloConverter.java`)

```
import javax.swing.*;     // Needed for Swing classes
import java.awt.event.*; // Needed for ActionListener Interface

/**
   The KiloConverter class displays a JFrame that
   lets the user enter a distance in kilometers. When
   the Calculate button is clicked, a dialog box is
   displayed with the distance converted to miles.
```

```
*/

public class KiloConverter extends JFrame
{
   private JPanel panel;             // To reference a panel
   private JLabel messageLabel;      // To reference a label
   private JTextField kiloTextField; // To reference a text field
   private JButton calcButton;       // To reference a button
   private final int WINDOW_WIDTH = 310;  // Window width
   private final int WINDOW_HEIGHT = 100; // Window height

   /**
      Constructor
   */

   public KiloConverter()
   {
      // Set the window title.
      setTitle("Kilometer Converter");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Build the panel and add it to the frame.
      buildPanel();

      // Add the panel to the frame's content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      The buildPanel method adds a label, a text field,
      and a button to a panel.
   */

   private void buildPanel()
   {
      // Create a label to display instructions.
      messageLabel = new JLabel("Enter a distance " +
                                "in kilometers");

      // Create a text field 10 characters wide.
      kiloTextField = new JTextField(10);

      // Create a button with the caption "Calculate".
      calcButton = new JButton("Calculate");

      // Add an action listener to the button.
      calcButton.addActionListener(new CalcButtonListener());

      // Create a JPanel object and let the panel
      // field reference it.
      panel = new JPanel();

      // Add the label, text field, and button
      // components to the panel.
      panel.add(messageLabel);
      panel.add(kiloTextField);
      panel.add(calcButton);
   }

   /**
      CalcButtonListener is an action listener class for
      the Calculate button.
   */

   private class CalcButtonListener implements ActionListener
   {
      /**
         The actionPerformed method executes when the user
         clicks on the Calculate button.
         @param e The event object.
      */

      public void actionPerformed(ActionEvent e)
      {
         final double CONVERSION = 0.6214;
         String input;  // To hold the user's input
         double miles;  // The number of miles

         // Get the text entered by the user into the
         // text field.
         input = kiloTextField.getText();

         // Convert the input to miles.
         miles = Double.parseDouble(input) * CONVERSION;

         // Display the result.
         JOptionPane.showMessageDialog(null, input +
                  " kilometers is " + miles + " miles.");
```

```
105       }
106    }
107 
108    /**
109       main method
110    */
111 
112    public static void main(String[] args)
113    {
114       new KiloConverter();
115    }
116 }
```

First, notice that we've added the `import java.awt.event.*;` statement in line 2. This is necessary for our program to use the `ActionListener` interface. Next, look at the following code in line 81:

```
private class CalcButtonListener implements ActionListener
```

This is the header for an inner class that we will use to create event listener objects. The name of this class is `CalcButtonListener` and it implements the `ActionListener` interface. We could have named the class anything we wanted to, but because it will handle the `JButton` component's action events, it must implement the `ActionListener` interface. The class has one method, `actionPerformed`, which is required by the `ActionListener` interface. The header for the `actionPerformed` method appears in line 89 as follows:

```
public void actionPerformed(ActionEvent e)
```

This method will be executed when the user clicks the `JButton` component. It has one parameter, `e`, which is an `ActionEvent` object. This parameter receives the event object that is passed to the method when it is called. Although we do not actually use the `e` parameter in this method, we still have to list it inside the method header's parentheses because it is required by the `ActionListener` interface.

The `actionPerformed` method declares a constant for the conversion factor in line 91, and two local variables in lines 92 and 93: `input`, a reference to a `String` object; and `miles`, a `double`. The following statement appears in line 97:

```
input = kiloTextField.getText();
```

All `JTextField` objects have a `getText` method that returns the text contained in the text field. This will be any value entered into the text field by the user. The value is returned as a string. So, this statement retrieves any value entered by the user into the text field and assigns it to `input`.

The following statement appears in line 100:

```
miles = Double.parseDouble(input) * CONVERSION;
```

This statement converts the value in `input` to a `double`, and then multiplies it by the constant `CONVERSION`, which is set to 0.6214. This will convert the number of kilometers entered by the user to miles. The result is stored in the `miles` variable. The method's last statement, in lines 103 and 104, uses `JOptionPane` to display a dialog box showing the distance converted to miles as follows:

```
JOptionPane.showMessageDialog(null, input +
         " kilometers is " + miles + " miles.");
```

Writing an action listener class is only part of the process of handling a `JButton` component's action events. We must also create an object from the class and then register the object with the `JButton` component. When we register the action listener object with the `JButton` component, we are creating a connection between the two objects.

`JButton` components have a method named `addActionListener`, which is used for registering action event listeners. In line 63, which is in the `buildPanel` method, the following statement creates a `CalcButtonListener` object and registers that object with the `calcButton` object:

```
calcButton.addActionListener(new CalcButtonListener());
```

You pass the address of an action listener object as the argument to the `addActionListener` method. This statement uses the expression `new CalcButtonListener()` to create an instance of the `CalcButtonListener` class. The address of that instance is then passed to the `addActionListener` method. Now, when the user clicks the Calculate button, the `CalcButtonListener` object's `actionPerformed` method will be executed.

**TIP:** Instead of the one statement in line 63, we could have written the following two statements:

```
CalcButtonListener listener = new CalcButtonListener();
calcButton.addActionListener(listener);
```

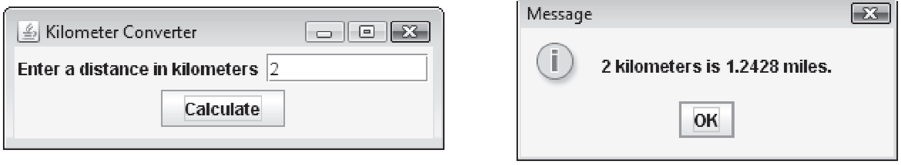
The first statement shown here declares a `CalcButtonListener` variable named `listener`, creates a new `CalcButtonListener` object, and assigns the object's address to the `listener` variable. The second statement passes the address in listener to the `addActionListener` method. These two statements accomplish the same thing as the one statement in line 63, but they declare a variable, `listener`, that we will not use again in the program. A better way is to use the one statement that appears in line 63 as follows:

```
calcButton.addActionListener(new CalcButtonListener());
```

Recall that the `new` key word creates an object and returns the object's address. This statement uses the `new` key word to create a `CalcButtonListener` object, and passes the object's address directly to the `addActionListener` method. Because we do not need to refer to the object again in the program, we do not assign the object's address to a variable. It is known as an *anonymous object*.

When this program is executed, the first window shown in Figure 23-10 is displayed on the screen. If the user enters 2 in the text field and clicks the Calculate button, the second window shown in the figure (a dialog box) appears. To exit the application, the user clicks the OK button on the dialog box, and then clicks the close button in the upper-right corner of the main window.

**Figure 23-10** Windows displayed by the `KiloConverter` class (Oracle Corporate Counsel)


This window appears first. The user enters 2 in the text field and then clicks the Calculate button.

Kilometer Converter

Enter a distance in kilometers: 2

This dialog box appears next.

Message

2 kilometers is 1.2428 miles.

## Background and Foreground Colors

Many of the Swing component classes have methods named `setBackground` and `setForeground`. You call these methods to change a component's color. The background color is the color of the component itself, and the foreground color is the color of text that might be displayed on the component.

The argument that you pass to the `setBackground` and `setForeground` methods is a color code. Table 23-3 lists several predefined constants that you can use for colors. To use these constants, you must have the `import java.awt.*;` statement in your code.

**Table 23-3** Color constants (Oracle Corporate Counsel)

| | |
|---|---|
| `Color.BLACK` | `Color.BLUE` |
| `Color.CYAN` | `Color.DARK_GRAY` |
| `Color.GRAY` | `Color.GREEN` |
| `Color.LIGHT_GRAY` | `Color.MAGENTA` |
| `Color.ORANGE` | `Color.PINK` |
| `Color.RED` | `Color.WHITE` |
| `Color.YELLOW` | |

For example, the following code creates a button with the text "OK" displayed on it. The `setBackground` and `setForeground` methods are called to make the button blue and the text yellow.

```
JButton okButton = new JButton("OK");
okButton.setBackground(Color.BLUE);
okButton.setForeground(Color.YELLOW);
```

The `ColorWindow` class in Code Listing 23-7 displays a window with a label and three buttons. When the user clicks a button, it changes the background color of the panel that contains the components and the foreground color of the label.

**Code Listing 23-7** (`ColorWindow.java`)

```
import javax.swing.*;    // Needed for Swing classes
import java.awt.*;       // Needed for Color class
import java.awt.event.*; // Needed for event listener interface

/**
   This class demonstrates how to set the background color of
   a panel and the foreground color of a label.
*/

public class ColorWindow extends JFrame
{
   private JLabel messageLabel;    // To display a message
   private JButton redButton;      // Changes color to red
   private JButton blueButton;     // Changes color to blue
   private JButton yellowButton;   // Changes color to yellow
   private JPanel panel;           // A panel to hold components
   private final int WINDOW_WIDTH = 200;  // Window width
   private final int WINDOW_HEIGHT = 125; // Window height

   /**
      Constructor
   */

   public ColorWindow()
   {
      // Set the title bar text.
      setTitle("Colors");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a label.
      messageLabel = new JLabel("Click a button to " +
                                "select a color.");

      // Create the three buttons.
      redButton = new JButton("Red");
      blueButton = new JButton("Blue");
      yellowButton = new JButton("Yellow");

      // Register an event listener with all 3 buttons.
      redButton.addActionListener(new RedButtonListener());
      blueButton.addActionListener(new BlueButtonListener());
      yellowButton.addActionListener(new YellowButtonListener());

      // Create a panel and add the components to it.
      panel = new JPanel();
      panel.add(messageLabel);
      panel.add(redButton);
      panel.add(blueButton);
      panel.add(yellowButton);

      // Add the panel to the content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      Private inner class that handles the event when
      the user clicks the Red button.
   */

   private class RedButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Set the panel's background to red.
         panel.setBackground(Color.RED);

         // Set the label's text to blue.
         messageLabel.setForeground(Color.BLUE);
      }
   }

   /**
      Private inner class that handles the event when
      the user clicks the Blue button.
   */

   private class BlueButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Set the panel's background to blue.
         panel.setBackground(Color.BLUE);
```

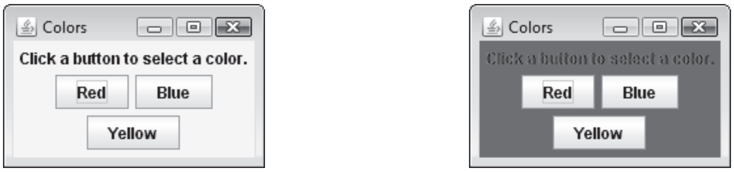
```
 91
 92          // Set the label's text to yellow.
 93          messageLabel.setForeground(Color.YELLOW);
 94       }
 95    }
 96
 97    /**
 98       Private inner class that handles the event when
 99       the user clicks the Yellow button.
100    */
101
102    private class YellowButtonListener implements ActionListener
103    {
104       public void actionPerformed(ActionEvent e)
105       {
106          // Set the panel's background to yellow.
107          panel.setBackground(Color.YELLOW);
108
109          // Set the label's text to black.
110          messageLabel.setForeground(Color.BLACK);
111       }
112    }
113
114    /**
115       main method
116    */
117
118    public static void main(String[] args)
119    {
120       new ColorWindow();
121    }
122 }
```

Notice that this class has three action listener classes, one for each button. The action listener classes are `RedButtonListener`, `BlueButtonListener`, and `YellowButtonListener`. The following statements, in lines 45 through 47, register instances of these classes with the appropriate button components:

```
redButton.addActionListener(new RedButtonListener());
blueButton.addActionListener(new BlueButtonListener());
yellowButton.addActionListener(new YellowButtonListener());
```

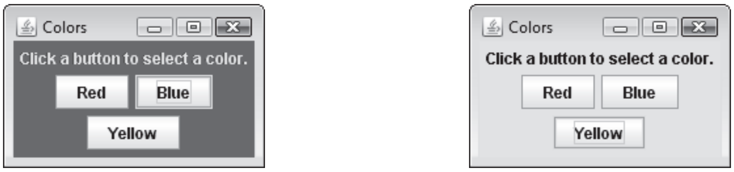
When you run the program, the window shown in Figure 23-11 appears.

**Figure 23-11** The window produced by the `ColorWindow` class (Oracle Corporate Counsel)


Colors
Click a button to select a color.
Red
Blue
Yellow
Colors
Click a button to select a color.
Red
Blue
Yellow

The window components first appear in their default colors.

When the user clicks on the Red button, the panel turns red and the label turns blue.


Colors
Click a button to select a color.
Red
Blue
Yellow
Colors
Click a button to select a color.
Red
Blue
Yellow

When the user clicks on the Blue button, the panel turns blue and the label turns yellow.

When the user clicks on the Yellow button, the panel turns yellow and the label turns black.

### Changing the Background Color of a `JFrame` Object's Content Pane

Recall that a `JFrame` object has a content pane, which is a container for all the components that are added to the `JFrame`. When you add a component to a `JFrame` object, you are actually adding it to the object's content pane. In the example shown in this section, we added a label and some buttons to a panel, and then added the panel to the `JFrame` object's content pane. When we changed the background color, we changed the background color of the panel. In this example, the color of the content pane does not matter because it is completely filled up by the panel. The color of the panel covers up the color of the content pane.

In some cases, where you have not filled up the `JFrame` object's content pane with a panel, you might want to change the background color of the content pane. If you wish to change the background color of a `JFrame` object's content pane, you must call the content pane's `setBackground` method, not the `JFrame` object's `setBackground` method. For example, in a class that extends the `JFrame` class, the following statement can be used to change the content pane's background to blue:

```
getContentPane().setBackground(Color.BLUE);
```

In this statement, the `getContentPane` method is called to get a reference to the `JFrame` object's content pane. This reference is then used to call the content pane's `setBackground` method. As a result, the content pane's background color will change to blue.

## The `ActionEvent` Object

The action listener's `actionPerformed` method has a parameter variable named `e` that is declared as follows:

```
ActionEvent e
```

`ActionEvent` is a class that is defined in the Java API. When an action event occurs, an object of the `ActionEvent` class is created, the action listener's `actionPerformed` method is called, and a reference to the `ActionEvent` object is passed into the `e` parameter variable. So, when the `actionPerformed` method executes, the `e` parameter references the event object that was generated in response to the event.

Earlier it was mentioned that the event object contains information about the event. If you wish, you can retrieve certain information about the event by calling one of the event object's methods. Two of the `ActionEvent` methods are listed in Table 23-4.

**Table 23-4** ActionEvent methods

| Method Name | Description |
|---|---|
| `getActionCommand()` | Returns the action command for this event as a `String` |
| `getSource()` | Returns a reference to the object that generated this event |

### The `getActionCommand` Method

The first method listed in Table 23-4, `getActionCommand`, returns the *action command* that is associated with the event. When a `JButton` component generates an event, the action command is the text that appears on the button. The `getActionCommand` returns this text as a `String`. You can use the `getActionCommand` method to determine which button was clicked when several buttons share the same action listener class.

To demonstrate, look at the `EventObjectWindow` class in Code Listing 23-8. It produces a window with three buttons. The buttons have the text "Button 1", "Button 2", and "Button 3". The action listener class displays the contents of the event object's action command when any of these buttons are clicked.

**Code Listing 23-8** `(EventObjectWindow.java)`

```
import javax.swing.*;    // Needed for Swing classes
import java.awt.event.*; // Needed for event listener interface

/**
    This class demonstrates how to retrieve the action command
    from an event object.
*/

public class EventObject extends JFrame
{
   private JButton button1;     // Button 1
   private JButton button2;     // Button 2
   private JButton button3;     // Button 3
   private JPanel panel;        // A panel to hold components
   private final int WINDOW_WIDTH = 300;  // Window width
   private final int WINDOW_HEIGHT = 70;  // Window height

   /**
         Constructor
   */

   public EventObject()
   {
      // Set the title bar text.
      setTitle("Event Object Demonstration");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the three buttons.
      button1 = new JButton("Button 1");
      button2 = new JButton("Button 2");
      button3 = new JButton("Button 3");

      // Register an event listener with all 3 buttons.
      button1.addActionListener(new ButtonListener());
      button2.addActionListener(new ButtonListener());
      button3.addActionListener(new ButtonListener());

      // Create a panel and add the buttons to it.
      panel = new JPanel();
      panel.add(button1);
      panel.add(button2);
      panel.add(button3);

      // Add the panel to the content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      Private inner class that handles the event when
      the user clicks a button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
```

```
64       {
65          // Get the action command.
66          String actionCommand = e.getActionCommand();
67
68          // Determine which button was clicked and display
69          // a message.
70          if (actionCommand.equals("Button 1"))
71          {
72             JOptionPane.showMessageDialog(null, "You clicked " +
73                                           "the first button.");
74          }
75          else if (actionCommand.equals("Button 2"))
76          {
77             JOptionPane.showMessageDialog(null, "You clicked " +
78                                           "the second button.");
79          }
80          else if (actionCommand.equals("Button 3"))
81          {
82             JOptionPane.showMessageDialog(null, "You clicked " +
83                                           "the third button.");
84          }
85       }
86    }
87
88    /**
89       main method
90    */
91
92    public static void main(String[] args)
93    {
94       new EventObject();
95    }
96 }
```

Previously you saw the `ColorWindow` class, in Code Listing 23-7, which had three buttons and three different action listener classes. The `EventObjectWindow` class also has three buttons, but only one action listener class. In lines 39 through 41, we create and register three separate instances of the class with the three buttons as follows:

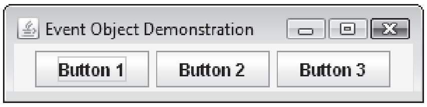
```
button1.addActionListener(new ButtonListener());
button2.addActionListener(new ButtonListener());
button3.addActionListener(new ButtonListener());
```

Figure 23-12 shows the output of the application when the user clicks Button 1, Button 2, and Button 3.

**Figure 23-12** Output of `EventObjectWindow` class (Oracle Corporate Counsel)

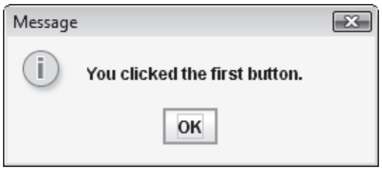
This window appears first.

1



The user clicks Button 1 and this dialog box appears next. The user clicks the OK button to dismiss the dialog box.

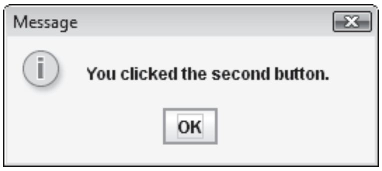
2



You clicked the first button.

The user clicks Button 2 and this dialog box appears next. The user clicks the OK button to dismiss the dialog box.

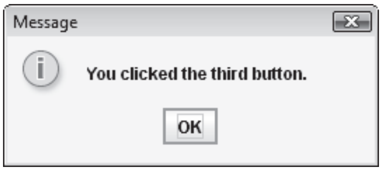
3



You clicked the second button.

The user clicks Button 3 and this dialog box appears next. The user clicks the OK button to dismiss the dialog box.

4



You clicked the third button.

**TIP:** The text that is displayed on a button is the default action command. You can change the action command by calling the `JButton` class's `setActionCommand` method. For example, assuming that `myButton` references a `JButton` component, the following statement would change the component's action command to "The button was clicked":

```
myButton.setActionCommand("The button was clicked");
```

**NOTE:** Changing a `JButton` component's action command does not change the text that is displayed on the button. For a demonstration of how to change the action command, see the *ActionCommand.java* file in this chapter's source code folder.

### The `getSource` Method

The second `ActionEvent` method listed in Table 23-4, `getSource`, returns a reference to the component that is the source of the event. As with the `getActionCommand` method, if you have several buttons and use objects of the same action listener class to respond to their events, you can use the `getSource` method to determine which button was clicked. For example, the `ButtonListener` class's `actionPerformed` method in Code Listing 23-8 could have been written as follows, to achieve the same result:

```
public void actionPerformed(ActionEvent e)
{
   // Determine which button was clicked and display
   // a message.
```

```
      if (e.getSource() == button1)
      {
         JOptionPane.showMessageDialog(null, "You clicked " +
                                       "the first button.");
      }
      else if (e.getSource() == button2)
      {
         JOptionPane.showMessageDialog(null, "You clicked " +
                                       "the second button.");
      }
      else if (e.getSource() == button3)
      {
         JOptionPane.showMessageDialog(null, "You clicked " +
                                       "the third button.");
      }
   }
```

See the *EventObjectWindow2.java* file in this chapter's source code folder for a demonstration of this code.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

23.1 What is a frame? How do you create a frame with Swing?

23.2 How do you set a frame's size?

23.3 How do you display a frame on the screen?

23.4 What is a content pane?

23.5 What is the difference between a frame and a panel?

23.6 What is an event listener?

23.7 If you are writing an event listener class for a `JButton` component, what interface must the class implement? What method must the class have? When is this method executed?

23.8 How do you register an event listener with a `JButton` component?

23.9 How do you change the background color of a component? How do you change the color of text displayed by a label or a button?

## 23.3 Layout Managers

**CONCEPT: A layout manager is an object that governs the positions and sizes of components in a container. The layout manager automatically repositions and, in some cases, resizes the components when the container is resized.**

An important part of designing a GUI application is determining the layout of the components that are displayed in the application's windows. The term *layout* refers to the positioning and sizing of components. In Java, you do not normally specify the exact location of a component within a window. Instead, you let a layout manager control the positions of components for you. A *layout manager* is an object that has its own rules about how components are to be positioned and sized, and it makes adjustments when necessary. For example, when the user resizes a window, the layout manager determines where the components should be moved to.

In order to use a layout manager with a group of components, you must place the components in a container, and then create a layout manager object. The layout manager object and the container work together. In this chapter we discuss the three layout managers described in Table 23-5. To use any of these classes, your code should have the following `import` statement: `import java.awt.*;`

**Table 23-5** Layout managers

| Layout Manager | Description |
|---|---|
| `FlowLayout` | Arranges components in rows; this is the default layout manager for `JPanel` objects |
| `BorderLayout` | Arranges components in five regions: north, south, east, west, and center; this is the default layout manager for a `JFrame` object's content pane |
| `GridLayout` | Arranges components in a grid with rows and columns |

## Adding a Layout Manager to a Container

You add a layout manager to a container, such as a content pane or a panel, by calling the `setLayout` method and passing a reference to a layout manager object as the argument. For example, the following code creates a `JPanel` object, then sets a `BorderLayout` object as its layout manager:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

Likewise, the following code might appear in the constructor of a class that extends the `JFrame` class. It sets a `FlowLayout` object as the layout manager for the content pane:

```
setLayout(new FlowLayout());
```

Once you establish a layout manager for a container, the layout manager governs the positions and sizes of the components that are added to the container.

## The `FlowLayout` Manager

The `FlowLayout` manager arranges components in rows. This is the default layout manager for `JPanel` objects. Here are some rules that the `FlowLayout` manager follows:

- You can add multiple components to a container that uses a `FlowLayout` manager.
- When you add components to a container that uses a `FlowLayout` manager, the components appear horizontally, from left to right, in the order that they were added to the component.
- When there is no more room in a row but more components are added, the new components "flow" to the next row.

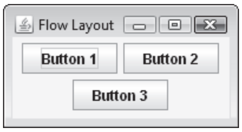
For example, the `FlowWindow` class shown in Code Listing 23-9 extends `JFrame`. This class creates a 200 pixel wide by 105 pixel high window. In the constructor, the `setLayout` method is called to give the content pane a `FlowLayout` manager. Then, three buttons are created and added to the content pane. The `main` method creates an instance of the `FlowWindow` class, which displays the window.

**Code Listing 23-9** `(FlowWindow.java)`

```
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for FlowLayout class

/**
   This class demonstrates how to use a FlowLayout manager
   with the content pane.
*/

public class FlowWindow extends JFrame
{
   private final int WINDOW_WIDTH = 200;  // Window width
   private final int WINDOW_HEIGHT = 105; // Window height

   /**
      Constructor
   */

   public FlowWindow()
   {
      // Set the title bar text.
      setTitle("Flow Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a FlowLayout manager to the content pane.
      setLayout(new FlowLayout());

      // Create three buttons.
      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      JButton button3 = new JButton("Button 3");

      // Add the three buttons to the content pane.
      add(button1);
      add(button2);
      add(button3);

      // Display the window.
      setVisible(true);
   }

   /**
      The main method creates an instance of the FlowWindow
      class, causing it to display its window.
   */

   public static void main(String[] args)
   {
      new FlowWindow();
   }
}
```

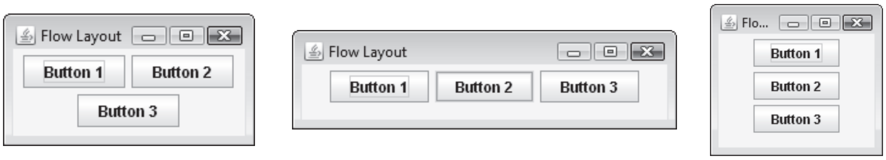
Figure 23-13 shows the window that is displayed by this class. Notice that the buttons appear from left to right in the order they were added to the content pane. Because there is only enough room for the first two buttons in the first row, the third button is positioned in the second row. By default, the content of each row is centered and there is a five pixel gap between the components.

**Figure 23-13** The window displayed by the `FlowWindow` class (Oracle Corporate Counsel)



If the user resizes the window, the layout manager repositions the components according to its rules. Figure 23-14 shows the appearance of the window in three different sizes.

**Figure 23-14** The arrangements of the buttons after resizing



### Adjusting the `FlowLayout` Alignment

The `FlowLayout` manager allows you to align components in the center of each row or along the left or right edge of each row. An overloaded constructor allows you to pass one of the following constants as an argument to set an alignment: `FlowLayout.CENTER`, `FlowLayout.LEFT`, or `FlowLayout.RIGHT`. Here is an example that sets left alignment:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

Figure 23-15 shows examples of windows that use a `FlowLayout` manager with left, center, and right alignment.

### Adjusting the `FlowLayout` Component Gaps

By default, the `FlowLayout` manager inserts a gap of five pixels between components, both horizontally and vertically. You can adjust this gap by passing values for the horizontal and vertical gaps as arguments to an overloaded `FlowLayout` constructor. The constructor has the following format:

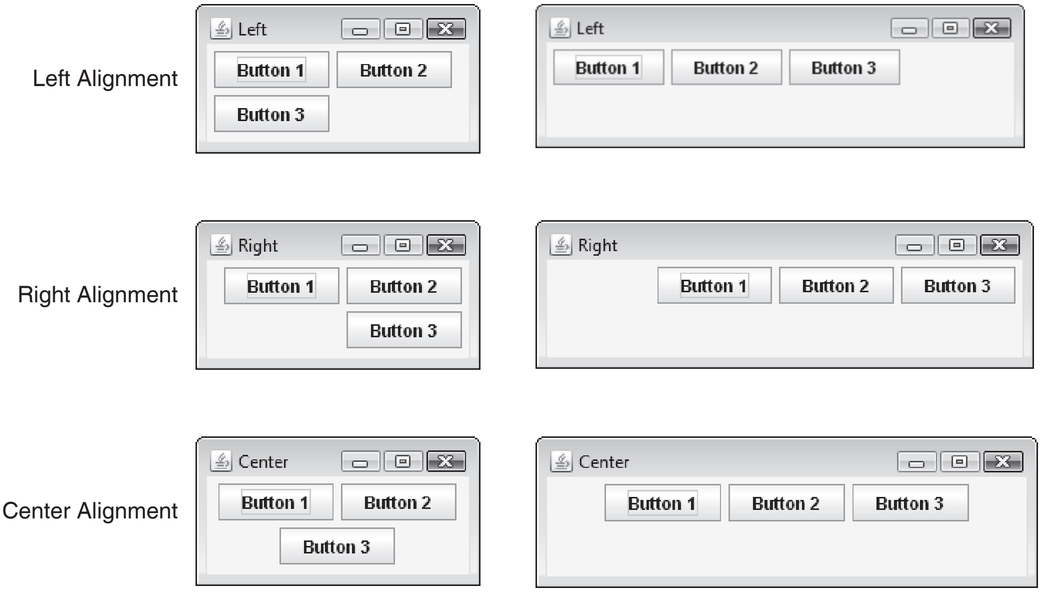
```
FlowLayout(int alignment, int horizontalGap, int verticalGap)
```

You pass one of the alignment constants discussed in the previous section to the `alignment` parameter. The `horizontalGap` parameter is the number of pixels to separate components horizontally, and the `verticalGap` parameter is the number of pixels to separate components vertically. Here is an example of the constructor call:

```
setLayout(new FlowLayout(FlowLayout.LEFT, 10, 7));
```

This statement causes components to be left aligned with a horizontal gap of 10 pixels and a vertical gap of seven pixels.

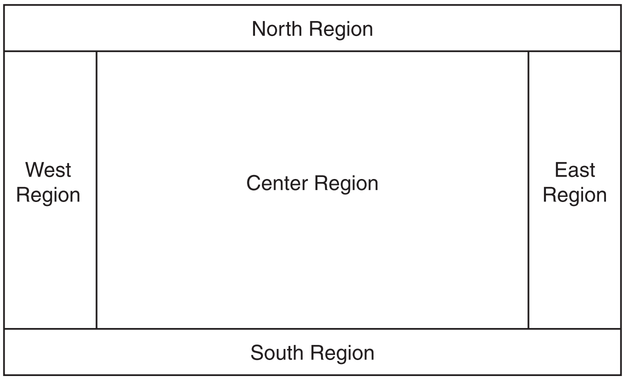
**Figure 23-15** Left, center, and right alignment (Oracle Corporate Counsel)


Left Alignment
Left
Button 1
Button 2
Button 3
Left
Button 1
Button 2
Button 3
Right Alignment
Right
Button 1
Button 2
Button 3
Right
Button 1
Button 2
Button 3
Center Alignment
Center
Button 1
Button 2
Button 3
Center
Button 1
Button 2
Button 3

## The `BorderLayout` Manager

The `BorderLayout` manager divides a container into five regions. The regions are known as north, south, east, west, and center. The arrangement of these regions is shown in Figure 23-16.

**Figure 23-16** The regions of a `BorderLayout` manager (Oracle Corporate Counsel)



When a component is placed into a container that is managed by a `BorderLayout` manager, the component must be placed into one of these five regions. Only one component at a time may be placed into a region. When adding a component to the container, you specify the region by passing one of the following constants as a second argument to the container's `add` method: `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, or `BorderLayout.CENTER`.

For example, look at the following code:

```
JPanel panel = new JPanel();
JButton button = new JButton("Click Me");
panel.setLayout(new BorderLayout());
panel.add(button, BorderLayout.NORTH);
```

The first statement creates a `JPanel` object, referenced by the `panel` variable. The second statement creates a `JButton` object, referenced by the `button` variable. The third statement sets the `JPanel` object's layout manager to a `BorderLayout` object. The fourth statement adds the `JButton` object to the `JPanel` object's north region.

If you do not pass a second argument to the `add` method, the component will be added to the center region. Here are some rules that the `BorderLayout` manager follows:

- Each region can hold only one component at a time.
- When a component is added to a region, the component is stretched so it fills up the entire region.

Look at the `BorderWindow` class shown in Code Listing 23-10, which extends `JFrame`. This class creates a 400 pixel wide by 300 pixel high window. In the constructor, the `setLayout` method is called to give the content pane a `BorderLayout` manager. Then, five buttons are created and each is added to a different region.

**Code Listing 23-10** `(BorderWindow.java)`

```
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for BorderLayout class
```

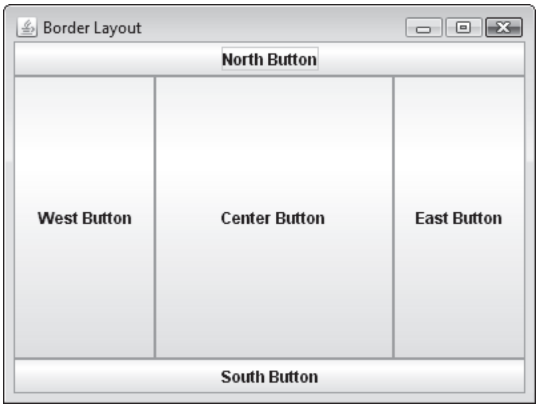
```

/**
   This class demonstrates the BorderLayout manager.
*/

public class BorderWindow extends JFrame
{
   private final int WINDOW_WIDTH = 400;   // Window width
   private final int WINDOW_HEIGHT = 300;  // Window height

   /**
      Constructor
   */

   public BorderWindow()
   {
      // Set the title bar text.
      setTitle("Border Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager to the content pane.
      setLayout(new BorderLayout());

      // Create five buttons.
      JButton button1 = new JButton("North Button");
      JButton button2 = new JButton("South Button");
      JButton button3 = new JButton("East Button");
      JButton button4 = new JButton("West Button");
      JButton button5 = new JButton("Center Button");

      // Add the five buttons to the content pane.
      add(button1, BorderLayout.NORTH);
      add(button2, BorderLayout.SOUTH);
      add(button3, BorderLayout.EAST);
      add(button4, BorderLayout.WEST);
      add(button5, BorderLayout.CENTER);

      // Display the window.
      setVisible(true);
   }

   /**
```

```
      The main method creates an instance of the BorderWindow
      class, causing it to display its window.
   */

   public static void main(String[] args)
   {
      new BorderWindow();
   }
}
```

**NOTE:** A `JFrame` object's content pane is automatically given a `BorderLayout` manager. We have explicitly added it in Code Listing 23-10 so it is clear that we are using a `BorderLayout` manager.

Figure 23-17 shows the window that is displayed. Normally the size of a button is just large enough to accommodate the text that is displayed on the button. Notice that the buttons displayed in this window did not retain their normal size. Instead, they were stretched to fill all of the space in their regions. If the user resizes the window, the sizes of the components will be changed as well. This is shown in Figure 23-18.

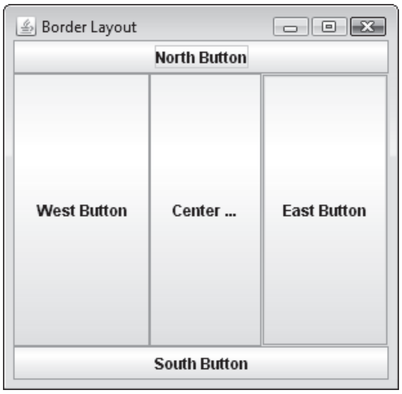
**Figure 23-17** The window displayed by the `BorderWindow` class (Oracle Corporate Counsel)



Here are the rules that govern how a `BorderLayout` manager resizes components:

- A component that is placed in the north or south regions may be resized horizontally so it fills up the entire region.
- A component that is placed in the east or west regions may be resized vertically so it fills up the entire region.

**Figure 23-18** The window resized (Oracle Corporate Counsel)



- A component that is placed in the center region may be resized both horizontally and vertically so it fills up the entire region.

**TIP:** You do not have to place a component in every region of a border layout. To achieve the desired positioning, you might want to place components in only a few of the layout regions. In Chapter 24, you will see examples of applications that do this.

By default there is no gap between the regions. You can use an overloaded version of the `BorderLayout` constructor to specify horizontal and vertical gaps, however. Here is the constructor's format:

```
BorderLayout(int horizontalGap, int verticalGap)
```

The `horizontalGap` parameter is the number of pixels to separate the regions horizontally, and the `verticalGap` parameter is the number of pixels to separate the regions vertically. Here is an example of the constructor call:

```
setLayout(new BorderLayout(5, 10));
```

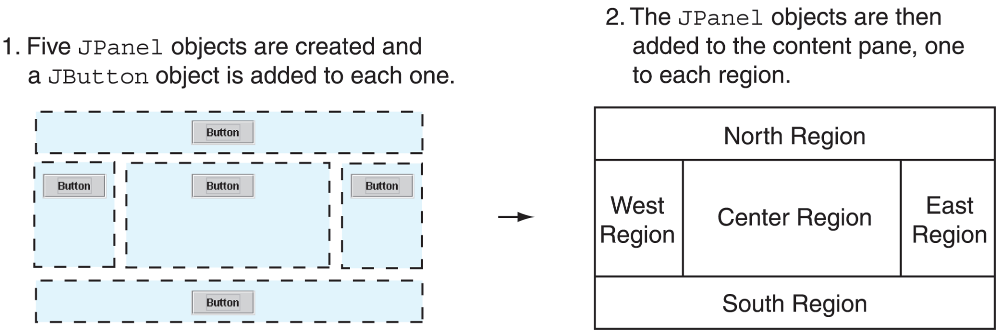
This statement causes the regions to appear with a horizontal gap of five pixels and a vertical gap of 10 pixels.

### Nesting Panels Inside a Container's Regions

You might think that the `BorderLayout` manager is limiting because it allows only one component per region, and the components that are placed in its regions are automatically resized to fill up any extra space. These limitations are easy to overcome, however, by adding components to panels and then nesting the panels inside the regions.

For example, suppose we wish to modify the `BorderWindow` class in Code Listing 23-10 so the buttons retain their original size. We can accomplish this by placing each button in a separate `JPanel` object and then adding the `JPanel` objects to the content pane's five regions. This is illustrated in Figure 23-19. As a result, the `BorderLayout` manager resizes the `JPanel` objects to fill up the space in the regions, not the buttons contained within the `JPanel` objects.

**Figure 23-19** Nesting `JPanel` objects inside each region



1. Five `JPanel` objects are created and a `JButton` object is added to each one.

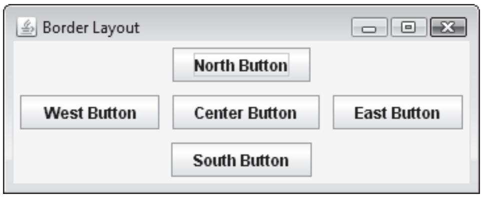
2. The `JPanel` objects are then added to the content pane, one to each region.

The `BorderPanelWindow` class in Code Listing 23-11 demonstrates this technique. This class also introduces a new way of sizing windows. Notice that the constructor does not explicitly set the size of the window with the `setSize` method. Instead, it calls the `pack` method just before calling the `setVisible` method. The `pack` method, which is inherited from `JFrame`, automatically sizes the window to accommodate the components contained within it. Figure 23-20 shows the window that the class displays.

**Code Listing 23-11** (`BorderPanelWindow.java`)

```java
import java.awt.*;     // Needed for BorderLayout class
import javax.swing.*;  // Needed for Swing classes

/**
   This class demonstrates how JPanels can be nested
   inside each region of a content pane governed by
   a BorderLayout manager.
*/

public class BorderPanelWindow extends JFrame
{
   /**
      Constructor
   */

   public BorderPanelWindow()
   {
      // Set the title bar text.
      setTitle("Border Layout");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager to the content pane.
      setLayout(new BorderLayout());

      // Create five panels.
      JPanel panel1 = new JPanel();
      JPanel panel2 = new JPanel();
      JPanel panel3 = new JPanel();
      JPanel panel4 = new JPanel();
      JPanel panel5 = new JPanel();

      // Create five buttons.
      JButton button1 = new JButton("North Button");
      JButton button2 = new JButton("South Button");
      JButton button3 = new JButton("East Button");
      JButton button4 = new JButton("West Button");
      JButton button5 = new JButton("Center Button");

      // Add the buttons to the panels.
      panel1.add(button1);
      panel2.add(button2);
      panel3.add(button3);
      panel4.add(button4);
      panel5.add(button5);

      // Add the five panels to the content pane.
      add(panel1, BorderLayout.NORTH);
      add(panel2, BorderLayout.SOUTH);
      add(panel3, BorderLayout.EAST);
      add(panel4, BorderLayout.WEST);
      add(panel5, BorderLayout.CENTER);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The main method creates an instance of the
      BorderPanelWindow class, causing it to display
      its window.
   */

   public static void main(String[] args)
   {
      new BorderPanelWindow();
   }
}
```

**Figure 23-20** Window displayed by the `BorderPanelWindow` class (Oracle Corporate Counsel)

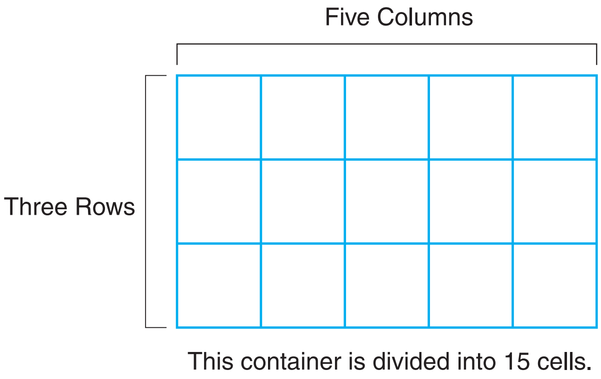

Border Layout
North Button
West Button
Center Button
East Button
South Button

**NOTE:** There are multiple layout managers at work in the `BorderPanelWindow` class. The content pane uses a `BorderLayout` manager, and each of the `JPanel` objects use a `FlowLayout` manager.

## The `GridLayout` Manager

The `GridLayout` manager creates a grid with rows and columns, much like a spreadsheet. As a result, the container that is managed by a `GridLayout` object is divided into equally sized cells. Figure 23-21 illustrates a container with three rows and five columns. This means that the container is divided into 15 cells.

**Figure 23-21** The `GridLayout` manager divides a container into cells


Five Columns
Three Rows
This container is divided into 15 cells.

Here are some rules that the `GridLayout` manager follows:

- Each cell can hold only one component.
- All of the cells are the same size. This is the size of the largest component placed within the layout.
- A component that is placed in a cell is automatically resized to fill up any extra space.

You pass the number of rows and columns that a container should have as arguments to the `GridLayout` constructor. Here is the general format of the constructor:

```
GridLayout(int rows, int columns)
```

Here is an example of the constructor call:

```
setLayout(new GridLayout(2, 3));
```

This statement gives the container two rows and three columns, for a total of six cells. You can pass 0 as an argument for the rows or the columns, but not both. Passing 0 for both arguments will cause an error.

When adding components to a container that is governed by the `GridLayout` manager, you cannot specify a cell. Instead, the components are assigned to cells in the order they are added. The first component added to the container is assigned to the first cell, which is in the upper-left corner. As other components are added, they are assigned to the remaining cells in the first row, from left to right. When the first row is filled up, components are assigned to the cells in the second row, and so forth.

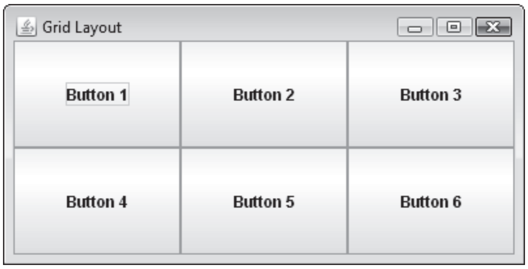
The `GridWindow` class shown in Code Listing 23-12 demonstrates. It creates a 400 pixel wide by 200 pixel high window, governed by a `GridLayout` manager. The content pane is divided into two rows and three columns, and a button is added to each cell. Figure 23-22 shows the window displayed by the class.

**Code Listing 23-12** (`GridWindow.java`)

```
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for GridLayout class

/**
   This class demonstrates the GridLayout manager.
*/

public class GridWindow extends JFrame
{
   private final int WINDOW_WIDTH = 400;   // Window width
   private final int WINDOW_HEIGHT = 200;  // Window height

   /**
      Constructor
   */

   public GridWindow()
   {
      // Set the title bar text.
      setTitle("Grid Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a GridLayout manager to the content pane.
      setLayout(new GridLayout(2, 3));

      // Create six buttons.
      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      JButton button3 = new JButton("Button 3");
      JButton button4 = new JButton("Button 4");
      JButton button5 = new JButton("Button 5");
      JButton button6 = new JButton("Button 6");

      // Add the six buttons to the content pane.
      add(button1);  // Goes into row 1, column 1
      add(button2);  // Goes into row 1, column 2
      add(button3);  // Goes into row 1, column 3
      add(button4);  // Goes into row 2, column 1
      add(button5);  // Goes into row 2, column 2
      add(button6);  // Goes into row 2, column 3

      // Display the window.
      setVisible(true);
   }

   /**
      The main method creates an instance of the GridWindow
      class, causing it to display its window.
   */

   public static void main(String[] args)
   {
      new GridWindow();
   }
}
```

As previously mentioned, the `GridLayout` manager limits each cell to only one component and resizes components to fill up all of the space in a cell. To get around these limitations you can nest panels inside the cells and add other components to the panels. For example, the `GridPanelWindow` class shown in Code Listing 23-13 is a modification of the `GridWindow` class. It creates six panels and adds a button and a label to each panel. These panels are then added to the content pane's cells. Figure 23-23 shows the window displayed by this class.

**Figure 23-22** Window displayed by the `GridWindow` class (Oracle Corporate Counsel)

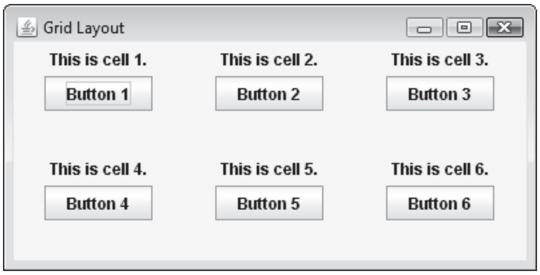


**Code Listing 23-13** (`GridPanelWindow.java`)

```java
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for GridLayout class

/**
   This class demonstrates how panels may be added to
   the cells created by a GridLayout manager.
*/

public class GridPanelWindow extends JFrame
{
   private final int WINDOW_WIDTH = 400;   // Window width
   private final int WINDOW_HEIGHT = 200;  // Window height

   /**
      Constructor
   */

   public GridPanelWindow()
   {
      // Set the title bar text.
      setTitle("Grid Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a GridLayout manager to the content pane.
      setLayout(new GridLayout(2, 3));

      // Create six buttons.
      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      JButton button3 = new JButton("Button 3");
      JButton button4 = new JButton("Button 4");
      JButton button5 = new JButton("Button 5");
      JButton button6 = new JButton("Button 6");

      // Create six labels.
      JLabel label1 = new JLabel("This is cell 1.");
      JLabel label2 = new JLabel("This is cell 2.");
      JLabel label3 = new JLabel("This is cell 3.");
      JLabel label4 = new JLabel("This is cell 4.");
      JLabel label5 = new JLabel("This is cell 5.");
      JLabel label6 = new JLabel("This is cell 6.");

      // Create six panels.
      JPanel panel1 = new JPanel();
      JPanel panel2 = new JPanel();
      JPanel panel3 = new JPanel();
      JPanel panel4 = new JPanel();
      JPanel panel5 = new JPanel();
      JPanel panel6 = new JPanel();

      // Add the labels to the panels.
      panel1.add(label1);
      panel2.add(label2);
      panel3.add(label3);
      panel4.add(label4);
      panel5.add(label5);
      panel6.add(label6);

      // Add the buttons to the panels.
      panel1.add(button1);
      panel2.add(button2);
      panel3.add(button3);
      panel4.add(button4);
      panel5.add(button5);
      panel6.add(button6);

      // Add the panels to the content pane.
      add(panel1);  // Goes into row 1, column 1
      add(panel2);  // Goes into row 1, column 2
      add(panel3);  // Goes into row 1, column 3
      add(panel4);  // Goes into row 2, column 1
      add(panel5);  // Goes into row 2, column 2
      add(panel6);  // Goes into row 2, column 3

      // Display the window.
      setVisible(true);
```

```
   }

   /**
      The main method creates an instance of the
      GridPanelWindow class, displaying its window.
   */

   public static void main(String[] args)
   {
      new GridPanelWindow();
   }
}
```

**Figure 23-23** Window displayed by the `GridPanelWindow` class (Oracle Corporate Counsel)



Because we have containers nested inside the content pane, there are multiple layout managers at work in the `GridPanelWindow` class. The content pane uses a `GridLayout` manager, and each of the `JPanel` objects uses a `FlowLayout` manager.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

23.10 How do you add a layout manager to a container?

23.11 Which layout manager divides a container into regions known as north, south, east, west, and center?

23.12 Which layout manager arranges components in a row, from left to right, in the order they were added to the container?

23.13 Which layout manager arranges components in rows and columns?

23.14 How many components can you have at one time in a `BorderLayout` region? In a `GridLayout` cell?

23.15 How do you prevent the `BorderLayout` manager from resizing a component that has been placed in its region?

23.16 How can you cause a content pane to be automatically sized to accommodate the components contained within it?

23.17 What is the default layout manager for a `JFrame` object's content pane? For a `JPanel` object?

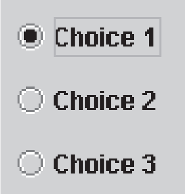
## 23.4 Radio Buttons and Check Boxes

**CONCEPT:** **Radio buttons normally appear in groups of two or more and allow the user to select one of several possible options. Check boxes, which may appear alone or in groups, allow the user to make yes/no or on/off selections.**

### Radio Buttons

*Radio buttons* are useful when you want the user to select one choice from several possible options. Figure 23-24 shows a group of radio buttons.

**Figure 23-24** Radio buttons (Oracle Corporate Counsel)



A radio button may be selected or deselected. Each radio button has a small circle that appears filled in when the radio button is selected and appears empty when the radio button is deselected. You use the `JRadioButton` class to create radio buttons. Here are the general formats of two `JRadioButton` constructors:

```
JRadioButton(String text)
JRadioButton(String text, boolean selected)
```

The first constructor shown creates a deselected radio button. The argument passed to the `text` parameter is the string that is displayed next to the radio button. For example, the following statement creates a radio button with the text "Choice 1" displayed next to it. The radio button initially appears deselected.

```
JRadioButton radio1 = new JRadioButton("Choice 1");
```

The second constructor takes an additional `boolean` argument, which is passed to the `selected` parameter. If `true` is passed as the `selected` argument, the radio button initially appears selected. If `false` is passed, the radio button initially appears deselected. For example, the following statement creates a radio button with the text "Choice 1" displayed next to it. The radio button initially appears selected.

```
JRadioButton radio1 = new JRadioButton("Choice 1", true);
```

Radio buttons are normally grouped together. When a set of radio buttons are grouped together, only one of the radio buttons in the group may be selected at any time. Clicking a radio button selects it and automatically deselects any other radio button in the same group. Because only one radio button in a group can be selected at any given time, the buttons are said to be *mutually exclusive*.

> **NOTE:** The name "radio button" refers to the old car radios that had push buttons for selecting stations. Only one of the buttons could be pushed in at a time. When you pushed a button in, it automatically popped out any other button that was pushed in.

### Grouping with the `ButtonGroup` class

Once you have created the `JRadioButton` objects that you wish to appear in a group, you must create an instance of the `ButtonGroup` class, and then add the `JRadioButton` objects to it. The `ButtonGroup` object creates the mutually exclusive relationship among the radio buttons that it contains. The following code shows an example:

```
// Create three radio buttons.
JRadioButton radio1 = new JRadioButton("Choice 1", true);
JRadioButton radio2 = new JRadioButton("Choice 2");
JRadioButton radio3 = new JRadioButton("Choice 3");

// Create a ButtonGroup object.
ButtonGroup group = new ButtonGroup();

// Add the radio buttons to the ButtonGroup object.
group.add(radio1);
group.add(radio2);
group.add(radio3);
```

Although you add radio buttons to a `ButtonGroup` object, `ButtonGroup` objects are not containers like `JPanel` objects, or content frames. The function of a `ButtonGroup` object is to deselect all the other radio buttons when one of them is selected. If you wish to add the radio buttons to a panel or a content frame, you must add them individually, as shown here:

```
// Add the radio buttons to the JPanel referenced by panel.
panel.add(radio1);
panel.add(radio2);
panel.add(radio3);
```

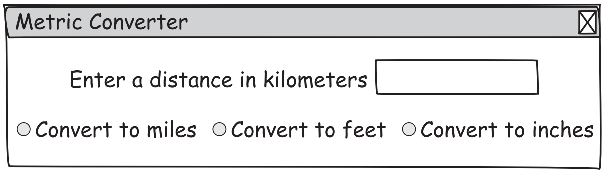
### Responding to Radio Button Events

Just like `JButton` objects, `JRadioButton` objects generate an action event when they are clicked. To respond to a radio button action event, you must write an action listener class and then register an instance of that class with the `JRadioButton` object. To demonstrate, we will look at the `MetricConverter` class, which is similar to the `KiloConverter` class shown earlier. The `MetricConverter` class presents a window in which the user can enter a distance in kilometers, and then click radio buttons to see that distance converted to miles, feet, or inches. The conversion formulas are as follows:

$$\mathit{Miles} = \mathit{Kilometers} \times 0.6214$$
$$\mathit{Feet} = \mathit{Kilometers} \times 3281.0$$
$$\mathit{Inches} = \mathit{Kilometers} \times 39370.0$$

Figure 23-25 shows a sketch of what the window will look like. As you can see from the sketch, the window will have a label, a text field, and three radio buttons. When the user clicks on one of the radio buttons, the distance will be converted to the selected units and displayed in a separate `JOptionPane` dialog box.

**Figure 23-25** Metric Converter window (Oracle Corporate Counsel)



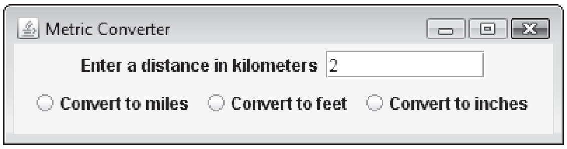
The `MetricConverter` class is shown in Code Listing 23-14. The class initially displays the window shown at the top of Figure 23-26. The figure also shows the dialog boxes that are displayed when the user clicks any of the radio buttons.

**Code Listing 23-14** `(MetricConverter.java)`

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
   The MetricConverter class lets the user enter a
   distance in kilometers. Radio buttons can be selected to
   convert the kilometers to miles, feet, or inches.
*/

public class MetricConverter extends JFrame
{
   private JPanel panel;                     // A holding panel
   private JLabel messageLabel;              // A message to the user
   private JTextField kiloTextField;         // To hold user input
   private JRadioButton milesButton;         // To convert to miles
   private JRadioButton feetButton;          // To convert to feet
   private JRadioButton inchesButton;        // To convert to inches
   private ButtonGroup radioButtonGroup;     // To group radio buttons
   private final int WINDOW_WIDTH = 400;     // Window width
   private final int WINDOW_HEIGHT = 100;    // Window height

   /**
      Constructor
   */

   public MetricConverter()
   {
      // Set the title.
      setTitle("Metric Converter");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Build the panel and add it to the frame.
      buildPanel();

      // Add the panel to the frame's content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      The buildPanel method adds a label, text field, and
      and three buttons to a panel.
   */

   private void buildPanel()
   {
      // Create the label, text field, and radio buttons.
      messageLabel = new JLabel("Enter a distance in kilometers");
      kiloTextField = new JTextField(10);
      milesButton = new JRadioButton("Convert to miles");
      feetButton = new JRadioButton("Convert to feet");
      inchesButton = new JRadioButton("Convert to inches");

      // Group the radio buttons.
      radioButtonGroup = new ButtonGroup();
      radioButtonGroup.add(milesButton);
      radioButtonGroup.add(feetButton);
      radioButtonGroup.add(inchesButton);

      // Add action listeners to the radio buttons.
      milesButton.addActionListener(new RadioButtonListener());
      feetButton.addActionListener(new RadioButtonListener());
      inchesButton.addActionListener(new RadioButtonListener());

      // Create a panel and add the components to it.
      panel = new JPanel();
      panel.add(messageLabel);
      panel.add(kiloTextField);
      panel.add(milesButton);
      panel.add(feetButton);
      panel.add(inchesButton);
   }

   /**
      Private inner class that handles the event when
      the user clicks one of the radio buttons.
   */

   private class RadioButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         String input;           // To hold the user's input
         String convertTo = "";  // The units we're converting to
         double result = 0.0;    // To hold the conversion

         // Get the kilometers entered.
         input = kiloTextField.getText();

         // Determine which radio button was clicked.
         if (e.getSource() == milesButton)
         {
            // Convert to miles.
            convertTo = " miles.";
            result = Double.parseDouble(input) * 0.6214;
         }
         else if (e.getSource() == feetButton)
         {
            // Convert to feet.
            convertTo = " feet.";
            result = Double.parseDouble(input) * 3281.0;
         }
         else if (e.getSource() == inchesButton)
         {
            // Convert to inches.
            convertTo = " inches.";
            result = Double.parseDouble(input) * 39370.0;
         }

         // Display the conversion.
         JOptionPane.showMessageDialog(null, input +
                     " kilometers is " + result + convertTo);
      }
   }

   /**
      The main method creates an instance of the
      MetricConverter class, displaying its window.
   */

   public static void main(String[] args)
   {
      new MetricConverter();
   }
}
```

**Figure 23-26** Window and dialog boxes displayed by the `MetricConverter` class (Oracle Corporate Counsel)

This window appears first. The user enters 2 into the text field.

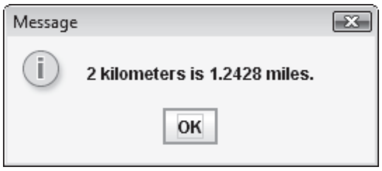


Metric Converter

Enter a distance in kilometers: 2

○ Convert to miles ○ Convert to feet ○ Convert to inches

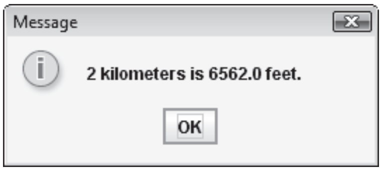
This dialog box appears when the user clicks the "Convert to miles" radio button.



Message

2 kilometers is 1.2428 miles.

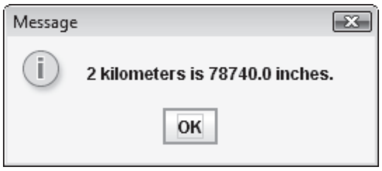
This dialog box appears when the user clicks the "Convert to feet" radio button.



Message

2 kilometers is 6562.0 feet.

This dialog box appears when the user clicks the "Convert to inches" radio button.



Message

2 kilometers is 78740.0 inches.

### Determining in Code Whether a Radio Button Is Selected

In many applications you will merely want to know whether a radio button is selected. The `JRadioButton` class's `isSelected` method returns a `boolean` value indicating whether the radio button is selected. If the radio button is selected, the method returns `true`. Otherwise, it returns `false`. In the following code, the `radio` variable references a radio button. The `if` statement calls the `isSelected` method to determine whether the radio button is selected.

```
if (radio.isSelected())
{
   // Code here executes if the radio
   // button is selected.
}
```

### Selecting a Radio Button in Code

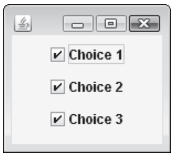
It is also possible to select a radio button in code with the `JRadioButton` class's `doClick` method. When the method is called, the radio button is selected just as if the user had clicked on it. As a result, an action event is generated. In the following statement, the `radio` variable references a radio button. When this statement executes, the radio button will be selected.

```
radio.doClick();
```

## Check Boxes

A *check box* appears as a small box with a label appearing next to it. The window shown in Figure 23-27 has three check boxes.

**Figure 23-27** Check boxes (Oracle Corporate Counsel)



☑ Choice 1
☑ Choice 2
☑ Choice 3

Like radio buttons, check boxes may be selected or deselected at run time. When a check box is selected, a small check mark appears inside the box. Although check boxes are often displayed in groups, they are not usually grouped in a `ButtonGroup` like radio buttons. This is because check boxes are not normally used to make mutually exclusive selections. Instead, the user is allowed to select any or all of the check boxes that are displayed in a group.

You create a check box with the `JCheckBox` class. Here are the general formats of two `JCheckBox` constructors:

```
JCheckBox(String text)
JCheckBox(String text, boolean selected)
```

The first constructor shown creates a deselected check box. The argument passed to the `text` parameter is the string that is displayed next to the check box. For example, the following statement creates a check box with the text "Macaroni" displayed next to it. The check box initially appears deselected.

```
JCheckBox check1 = new JCheckBox("Macaroni");
```

The second constructor takes an additional `boolean` argument, which is passed to the `selected` parameter. If `true` is passed as the `selected` argument, the radio check box initially appears selected. If `false` is passed, the check box initially appears deselected. For example, the following statement creates a check box with the text “Macaroni” displayed next to it. The radio check box initially appears selected.

```
JCheckBox check1 = new JCheckBox("Macaroni", true);
```

### Responding to Check Box Events

When a `JCheckBox` object is selected or deselected, it generates an *item event*. You handle item events in a manner similar to the way you handle the action events that are generated by `JButton` and `JRadioButton` objects. First, you write an *item listener* class, which must meet the following requirements:

- It must implement the `ItemListener` interface.
- It must have a method named `itemStateChanged` with the following header:

```
    public void itemStateChanged(ItemEvent e)
```

**NOTE:** When implementing the `ItemListener` interface, your code must have the following import statement: `import java.awt.event.*;`

Once you have written an item listener class, you create an object of that class, and then register the item listener object with the `JCheckBox` component. When a `JCheckBox` component generates an event, it automatically executes the `itemStateChanged` method of the item listener object that is registered to it, passing the event object as an argument.

### Determining in Code Whether a Check Box Is Selected

As with `JRadioButton`, you use the `isSelected` method to determine whether a `JCheckBox` component is selected. The method returns a `boolean` value. If the check box is selected, the method returns `true`. Otherwise, it returns `false`. In the following code, the `checkBox` variable references a `JCheckBox` component. The `if` statement calls the `isSelected` method to determine whether the check box is selected.

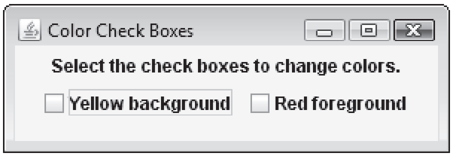
```
if (checkBox.isSelected())
{
   // Code here executes if the check
   // box is selected.
}
```

The `ColorCheckBoxWindow` class, shown in Code Listing 23-15, demonstrates how check boxes are used. It displays the window shown in Figure 23-28. When the “Yellow background” check box is selected, the background color of the content pane, the label, and the check boxes turns yellow. When this check box is deselected, the background colors go back to light gray. When the “Red foreground” check box is selected, the color of the text displayed in the label and the check boxes turns red. When this check box is deselected, the foreground colors go back to black.

**Code Listing 23-15** (ColorCheckBoxWindow.java)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   The ColorCheckBoxWindow class demonstrates how check boxes
   can be used.
*/

public class ColorCheckBoxWindow extends JFrame
{
   private JLabel messageLabel;        // A message to the user
   private JCheckBox yellowCheckBox;   // To select yellow background
   private JCheckBox redCheckBox;      // To select red foreground
   private final int WINDOW_WIDTH = 300;     // Window width
   private final int WINDOW_HEIGHT = 100;    // Window height

   /**
      Constructor
   */

   public ColorCheckBoxWindow()
   {
      // Set the text for the title bar.
      setTitle("Color Check Boxes");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a label.
      messageLabel = new JLabel("Select the check " +
                                "boxes to change colors.");

      // Create the check boxes.
      yellowCheckBox = new JCheckBox("Yellow background");
      redCheckBox = new JCheckBox("Red foreground");

      // Add an item listener to the check boxes.
      yellowCheckBox.addItemListener(new CheckBoxListener());
      redCheckBox.addItemListener(new CheckBoxListener());

      // Add a FlowLayout manager to the content pane.
      setLayout(new FlowLayout());

      // Add the label and check boxes to the content pane.
      add(messageLabel);
      add(yellowCheckBox);
      add(redCheckBox);

      // Display the window.
      setVisible(true);
   }

   /**
      Private inner class that handles the event when
      the user clicks one of the check boxes.
   */

   private class CheckBoxListener implements ItemListener
   {
      public void itemStateChanged(ItemEvent e)
      {
         // Determine which check box was clicked.
         if (e.getSource() == yellowCheckBox)
         {
            // Is the yellow check box selected? If so, we
            // want to set the background color to yellow.
            if (yellowCheckBox.isSelected())
            {
               // The yellow check box was selected. Set
               // the background color for the content
               // pane and the two check boxes to yellow.
               getContentPane().setBackground(Color.YELLOW);
               yellowCheckBox.setBackground(Color.YELLOW);
               redCheckBox.setBackground(Color.YELLOW);
            }
            else
            {
               // The yellow check box was deselected. Set
               // the background color for the content
               // pane and the two check boxes to light gray.
               getContentPane().setBackground(Color.LIGHT_GRAY);
               yellowCheckBox.setBackground(Color.LIGHT_GRAY);
               redCheckBox.setBackground(Color.LIGHT_GRAY);
            }
         }
         else if (e.getSource() == redCheckBox)
         {
            // Is the red check box selected? If so, we want
            // to set the foreground color to red.
            if (redCheckBox.isSelected())
            {
               // The red check box was selected. Set the
               // foreground color for the label and the
               // two check boxes to red.
               messageLabel.setForeground(Color.RED);
               yellowCheckBox.setForeground(Color.RED);
               redCheckBox.setForeground(Color.RED);
            }
            else
            {
               // The red check box was deselected. Set the
               // foreground color for the label and the
               // two check boxes to black.
               messageLabel.setForeground(Color.BLACK);
               yellowCheckBox.setForeground(Color.BLACK);
               redCheckBox.setForeground(Color.BLACK);
            }
         }
      }
   }

   /**
      The main method creates an instance of the
      ColorCheckBoxWindow class, displaying its window.
   */

   public static void main(String[] args)
   {
      new ColorCheckBoxWindow();
   }
}
```

**Figure 23-28** Window displayed by the `ColorCheckBoxWindow` class (Oracle Corporate Counsel)



Select the check boxes to change colors.

☐ Yellow background ☐ Red foreground

## Selecting a Check Box in Code

As with radio buttons, it is possible to select check boxes in code with the `JCheckBox` class's `doClick` method. When the method is called, the radio check box is selected just as if the user had clicked on it. As a result, an item event is generated. In the following statement, the `checkBox` variable references a `JCheckBox` object. When this statement executes, the check box will be selected.

```
checkBox.doClick();
```

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

23.18 You want the user to be able to select only one item from a group of items. Which type of component would you use for the items, radio buttons or check boxes?

23.19 You want the user to be able to select any number of items from a group of items. Which type of component would you use for the items, radio buttons or check boxes?

23.20 What is the purpose of a `ButtonGroup` object?

23.21 Do you normally add radio buttons, check boxes, or both to a `ButtonGroup` object?

23.22 What type of event does a radio button generate when the user clicks on it?

23.23 What type of event does a check box generate when the user clicks on it?

23.24 How do you determine in code whether a radio button is selected?

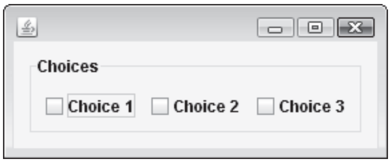
23.25 How do you determine in code whether a check box is selected?

## 23.5 Borders

**CONCEPT: A component can appear with several different styles of borders around it. A `Border` object specifies the details of a border. You use the `BorderFactory` class to create `Border` objects.**

Sometimes it is helpful to place a border around a component or a group of components on a panel. You can give windows a more organized look by grouping related components inside borders. For example, Figure 23-29 shows a group of check boxes that are enclosed in a border. In addition, notice that the border has a title.

**Figure 23-29** A group of check boxes with a titled border



`JPanel` components have a method named `setBorder`, which is used to add a border to the panel. The `setBorder` method accepts a `Border` object as its argument. A `Border` object contains detailed information describing the appearance of a border.

Rather than creating `Border` objects yourself, you should use the `BorderFactory` class to create them for you. The `BorderFactory` class has methods that return various types of borders. Table 23-6 describes borders that can be created with the `BorderFactory` class. The table also lists the `BorderFactory` methods that can be called to create the borders. Note that there are several overloaded versions of each method.

**NOTE:** If you use the `BorderFactory` class in your code, you should have the following `import` statement: `import javax.swing.*;`

**Table 23-6** Borders produced by the `BorderFactory` class

| Border | BorderFactory Method | Description |
|---|---|---|
| Compound border | `createCompoundBorder` | A border that has two parts: an inside edge and an outside edge. The inside and outside edges can be any of the other borders. |
| Empty border | `createEmptyBorder` | A border that contains only empty space. |
| Etched border | `createEtchedBorder` | A border with a 3-D appearance that looks "etched" into the background. |
| Line border | `createLineBorder` | A border that appears as a line. |
| Lowered bevel border | `createLoweredBevelBorder` | A border that looks like beveled edges. It has a 3-D appearance that gives the illusion of being sunken into the surrounding background. |
| Matte border | `createMatteBorder` | A line border that can have edges of different thicknesses. |
| Raised bevel border | `createRaisedBevelBorder` | A border that looks like beveled edges. It has a 3-D appearance that gives the illusion of being raised above the surrounding background. |
| Titled border | `createTitledBorder` | An etched border with a title. |

In this chapter, we will concentrate on empty borders, line borders, and titled borders.

### Empty Borders

An empty border is simply empty space around the edges of a component. To create an empty border, call the `BorderFactory` class's `createEmtpyBorder` method. Here is the method's general format:

```
BorderFactory.createEmptyBorder(int top, int left,
                                int bottom, int right);
```

The arguments passed into `top`, `left`, `bottom`, and `right` specify in pixels the size of the border's top, left, bottom, and right edges. The method returns a reference to a `Border` object. The following is an example of a statement that uses the method. Assume that the `panel` variable references a `JPanel` object.

```
panel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
```

After this statement executes, the `JPanel` referenced by `panel` will have an empty border of five pixels around each edge.

> **NOTE:** In case you've skipped ahead to this chapter, the `BorderFactory` methods are static, which means that you call them without creating an instance of the `BorderFactory` class. (You simply write *BorderFactory*. before the method name to call the method.) This is similar to the way the `Math` class and wrapper class methods we have discussed are called. Static methods are covered in Chapter 8.

### Line Borders

A line border is a line of a specified color and thickness that appears around the edges of a component. To create a line border, call the `BorderFactory` class's `createLineBorder` method. Here is the method's general format:

```
BorderFactory.createLineBorder(Color color, int thickness);
```

The arguments passed into `color` and `thickness` specify the color of the line and the size of the line in pixels. The method returns a reference to a `Border` object. The following is an example of a statement that uses the method. Assume that the `panel` variable references a `JPanel` object.

```
panel.setBorder(BorderFactory.createLineBorder(Color.RED, 1));
```

After this statement executes, the `JPanel` referenced by `panel` will have a red line border that is one pixel thick around its edges.

### Titled Borders

A titled border is an etched border with a title displayed on it. To create a titled border, call the `BorderFactory` class's `createTitledBorder` method. Here is the method's general format:

```
BorderFactory.createTitledBorder(String title);
```

The argument passed into `title` is the text that is to be displayed as the border's title. The method returns a reference to a `Border` object. The following is an example of a statement that uses the method. Assume that the `panel` variable references a `JPanel` object.

```
panel.setBorder(BorderFactory.createTitledBorder("Choices"));
```

After this statement executes, the `JPanel` referenced by `panel` will have an etched border with the title "Choices" displayed on it.

### Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

23.26 What method do you use to set a border around a component?

23.27 What is the preferred way of creating a `Border` object?

## 23.6 Focus on Problem Solving: Extending Classes from `JPanel`

**CONCEPT:** **By writing a class that is extended from the `JPanel` class, you can create a custom panel component that can hold other components and their related code.**

In the applications that you have studied so far in this chapter, we have used the `extends JFrame` clause in the class header to extend the class from the `JFrame` class. Recall that the extended class is then a specialized version of the `JFrame` class, and we use its constructor to create the panels, buttons, and all of the other components needed. This approach works well for simple applications. But for applications that use many components, this approach can be cumbersome. Bundling all of the code and event listeners for a large number of components into a single class can lead to a large and complex class. A better approach is to encapsulate smaller groups of related components and their event listeners into their own classes.

A commonly used technique is to extend a class from the `JPanel` class. This allows you to create your own specialized panel component, which can contain other components and related code such as event listeners. A complex application that uses numerous components can be constructed from several specialized panel components. In this section we will examine such an application.

### The Brandi's Bagel House Application

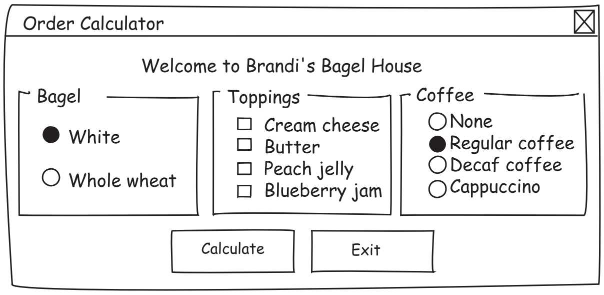
Brandi's Bagel House has a bagel and coffee delivery service for the businesses in her neighborhood. Customers may call in and order white and whole wheat bagels with a variety of toppings. In addition, customers may order three different types of coffee. (Delivery for coffee alone is not available, however.) Here is a complete price list:

| | |
|---|---|
| *Bagels:* | *White bagel $1.25, whole wheat bagel $1.50* |
| *Toppings:* | *Cream cheese $0.50, butter $0.25, peach jelly $0.75, blueberry jam $0.75* |
| *Coffee:* | *Regular coffee $1.25, decaf coffee $1.25, cappuccino $2.00* |

Brandi, the owner, needs an "order calculator" application that her staff can use to calculate the price of an order as it is called in. The application should display the subtotal, the amount of a 6 percent sales tax, and the total of the order. Figure 23-30 shows a sketch of the application's window. The user selects the type of bagel, toppings, and coffee, then clicks the Calculate button. A dialog box appears displaying the subtotal, amount of sales tax, and total. The user can exit the application by clicking either the Exit button or the standard close button in the upper-right corner.

The layout shown in the sketch can be achieved using a `BorderLayout` manager with the window's content pane. The label that displays "Welcome to Brandi's Bagel House" is in the north region, the radio buttons for the bagel types are in the west region, the check boxes for the toppings are in the center region, the radio buttons for the coffee selection are in the east region, and the Calculate and Exit buttons are in the south region. To construct this window, we create the following specialized panel classes that are extended from `JPanel`:

**Figure 23-30** Sketch of the Order Calculator window (Oracle Corporate Counsel)


Order Calculator
☒
Welcome to Brandi's Bagel House
Bagel
● White
○ Whole wheat
Toppings
☐ Cream cheese
☐ Butter
☐ Peach jelly
☐ Blueberry jam
Coffee
○ None
● Regular coffee
○ Decaf coffee
○ Cappuccino
Calculate
Exit

- **`GreetingsPanel`**. This panel contains the label that appears in the window's north region.
- **`BagelPanel`**. This panel contains the radio buttons for the types of bagels.
- **`ToppingPanel`**. This panel contains the check boxes for the types of bagels.
- **`CoffeePanel`**. This panel contains the radio buttons for the coffee selections.

(We will not create a specialized panel for the Calculate and Exit buttons. The reason is explained later.) After these classes have been created, we can create objects from them and add the objects to the correct regions of the window's content pane. Let's take a closer look at each of these classes.

## The `GreetingPanel` Class

The `GreetingPanel` class holds the label displaying the text "Welcome to Brandi's Bagel House". Code Listing 23-16 shows the class, which extends `JPanel`.

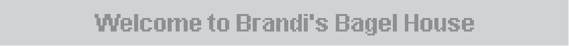
**Code Listing 23-16** (`GreetingPanel.java`)

```java
import javax.swing.*;

/**
   The GreetingPanel class displays a greeting in a panel.
*/

public class GreetingPanel extends JPanel
{
   private JLabel greeting; // To display a greeting

   /**
      Constructor
   */

   public GreetingPanel()
   {
      // Create the label.
18         greeting = new JLabel("Welcome to Brandi's Bagel House");
19 
20         // Add the label to this panel.
21         add(greeting);
22      }
23   }
```

In line 21 the `add` method is called to add the `JLabel` component referenced by `greeting`. Notice that we are calling the method without an object reference and a dot preceding it. This is because the method was inherited from the `JPanel` class, and we can call it just as if it were written into the `GreetingPanel` class declaration.

When we create an instance of this class, we are creating a `JPanel` component that displays a label with the text "Welcome to Brandi's Bagel House". Figure 23-31 shows how the component will appear when it is placed in the window's north region.

**Figure 23-31** Appearance of the `GreetingPanel` component



Welcome to Brandi's Bagel House

## The `BagelPanel` Class

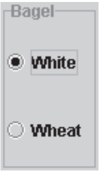
The `BagelPanel` class holds the radio buttons for the types of bagels. Notice that this panel uses a `GridLayout` manager with two rows and one column. Code Listing 23-17 shows the class, which is extended from `JPanel`.

**Code Listing 23-17** (`BagelPanel.java`)

```
import javax.swing.*;
import java.awt.*;

/**
   The BagelPanel class allows the user to select either
   a white or whole wheat bagel.
*/

public class BagelPanel extends JPanel
{
   // The following constants are used to indicate
   // the cost of each type of bagel.
   public final double WHITE_BAGEL = 1.25;
   public final double WHEAT_BAGEL = 1.50;

   private JRadioButton whiteBagel;  // To select white
   private JRadioButton wheatBagel;  // To select wheat
   private ButtonGroup bg;           // Radio button group

   /**
      Constructor
   */

   public BagelPanel()
   {
      // Create a GridLayout manager with
      // two rows and one column.
      setLayout(new GridLayout(2, 1));

      // Create the radio buttons.
      whiteBagel = new JRadioButton("White", true);
      wheatBagel = new JRadioButton("Wheat");

      // Group the radio buttons.
      bg = new ButtonGroup();
      bg.add(whiteBagel);
      bg.add(wheatBagel);

      // Add a border around the panel.
      setBorder(BorderFactory.createTitledBorder("Bagel"));

      // Add the radio buttons to the panel.
      add(whiteBagel);
      add(wheatBagel);
   }

   /**
      getBagelCost method
      @return The cost of the selected bagel.
   */

   public double getBagelCost()
   {
      double bagelCost = 0.0;

      if (whiteBagel.isSelected())
         bagelCost = WHITE_BAGEL;
      else
         bagelCost = WHEAT_BAGEL;

      return bagelCost;
   }
}
```

Notice that the `whiteBagel` radio button is automatically selected when it is created. This is the default choice. This class does not have an inner event listener class because we do not want to execute any code when the user selects a bagel. Instead, we want this class to be able to report the cost of the selected bagel. That is the purpose of the `getBagelCost` method, which returns the cost of the selected bagel as a `double`. (This method will be called by the Calculate button's event listener.) Figure 23-32 shows how the component appears when it is placed in the window's west region.

**Figure 23-32** Appearance of the `BagelPanel` component (Oracle Corporate Counsel)



Bagel

● White

○ Wheat

## The `ToppingPanel` Class

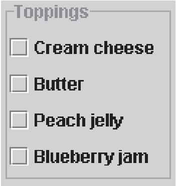
The `ToppingPanel` class holds the check boxes for the available toppings. Code Listing 23-18 shows the class, which is also extended from `JPanel`.

**Code Listing 23-18** (`ToppingPanel.java`)

```java
import javax.swing.*;
import java.awt.*;

/**
   The ToppingPanel class allows the user to select
   the toppings for the bagel.
*/

public class ToppingPanel extends JPanel
{
   // The following constants are used to indicate
   // the cost of toppings.
   public final double CREAM_CHEESE = 0.50;
   public final double BUTTER = 0.25;
   public final double PEACH_JELLY = 0.75;
   public final double BLUEBERRY_JAM = 0.75;

   private JCheckBox creamCheese;       // To select cream cheese
   private JCheckBox butter;            // To select butter
   private JCheckBox peachJelly;        // To select peach jelly
   private JCheckBox blueberryJam;      // To select blueberry jam

   /**
      Constructor
   */

   public ToppingPanel()
   {
      // Create a GridLayout manager with
      // four rows and one column.
      setLayout(new GridLayout(4, 1));

      // Create the check boxes.
      creamCheese = new JCheckBox("Cream cheese");
      butter = new JCheckBox("Butter");
      peachJelly = new JCheckBox("Peach jelly");
      blueberryJam = new JCheckBox("Blueberry jam");

      // Add a border around the panel.
      setBorder(BorderFactory.createTitledBorder("Toppings"));

      // Add the check boxes to the panel.
      add(creamCheese);
      add(butter);
      add(peachJelly);
      add(blueberryJam);
   }

   /**
      getToppingCost method
      @return The cost of the selected toppings.
   */

   public double getToppingCost()
   {
      double toppingCost = 0.0;

      if (creamCheese.isSelected())
         toppingCost += CREAM_CHEESE;
      if (butter.isSelected())
         toppingCost += BUTTER;
      if (peachJelly.isSelected())
         toppingCost += PEACH_JELLY;
      if (blueberryJam.isSelected())
         toppingCost += BLUEBERRY_JAM;

      return toppingCost;
   }
}
```

As with the `BagelPanel` class, this class does not have an inner event listener class because we do not want to execute any code when the user selects a topping. Instead, we want this class to be able to report the total cost of all the selected toppings. That is the purpose of the `getToppingCost` method, which returns the cost of all the selected toppings as a `double`. (This method will be called by the Calculate button's event listener.) Figure 23-33 shows how the component appears when it is placed in the window's center region.

**Figure 23-33** Appearance of the `ToppingPanel` component (Oracle Corporate Counsel)



## The `CoffeePanel` Class

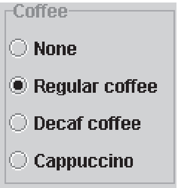
The `CoffeePanel` class holds the radio buttons for the available coffee selections. Code Listing 23-19 shows the class, which extends `JPanel`.

**Code Listing 23-19** (`CoffeePanel.java`)

```
import javax.swing.*;
import java.awt.*;

/**
   The CoffeePanel class allows the user to select coffee.
*/

public class CoffeePanel extends JPanel
{
   // The following constants are used to indicate
   // the cost of coffee.
   public final double NO_COFFEE = 0.0;
   public final double REGULAR_COFFEE = 1.25;
   public final double DECAF_COFFEE = 1.25;
   public final double CAPPUCCINO = 2.00;

   private JRadioButton noCoffee;       // To select no coffee
   private JRadioButton regularCoffee;  // To select regular coffee
   private JRadioButton decafCoffee;    // To select decaf
   private JRadioButton cappuccino;     // To select cappuccino
   private ButtonGroup bg;              // Radio button group

   /**
      Constructor
   */

   public CoffeePanel()
   {
      // Create a GridLayout manager with
      // four rows and one column.
      setLayout(new GridLayout(4, 1));

      // Create the radio buttons.
      noCoffee = new JRadioButton("None");
      regularCoffee = new JRadioButton("Regular coffee", true);
      decafCoffee = new JRadioButton("Decaf coffee");
      cappuccino = new JRadioButton("Cappuccino");

      // Group the radio buttons.
      bg = new ButtonGroup();
      bg.add(noCoffee);
      bg.add(regularCoffee);
      bg.add(decafCoffee);
      bg.add(cappuccino);

      // Add a border around the panel.
      setBorder(BorderFactory.createTitledBorder("Coffee"));

      // Add the radio buttons to the panel.
      add(noCoffee);
      add(regularCoffee);
      add(decafCoffee);
      add(cappuccino);
   }

   /**
      getCoffeeCost method
      @return The cost of the selected coffee.
   */

   public double getCoffeeCost()
   {
      double coffeeCost = 0.0;

      if (noCoffee.isSelected())
         coffeeCost = NO_COFFEE;
      else if (regularCoffee.isSelected())
         coffeeCost = REGULAR_COFFEE;
      else if (decafCoffee.isSelected())
         coffeeCost = DECAF_COFFEE;
      else if (cappuccino.isSelected())
         coffeeCost = CAPPUCCINO;

         return coffeeCost;
      }
   }
```

As with the `BagelPanel` and `ToppingPanel` classes, this class does not have an inner event listener class because we do not want to execute any code when the user selects coffee. Instead, we want this class to be able to report the cost of the selected coffee. The `getCoffeeCost` method returns the cost of the selected coffee as a `double`. (This method will be called by the Calculate button's event listener.) Figure 23-34 shows how the component appears when it is placed in the window's east region.

**Figure 23-34** Appearance of the `CoffeePanel` component

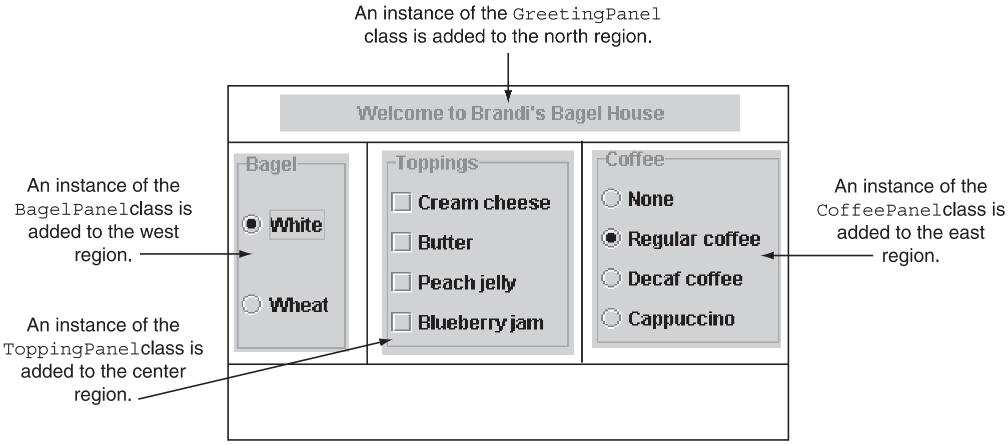


○ None
● Regular coffee
○ Decaf coffee
○ Cappuccino

## Putting It All Together

The last step in creating this application is to write a class that builds the application's window and adds the Calculate and Exit buttons. This class, which we name `OrderCalculatorGUI`, is extended from `JFrame` and uses a `BorderLayout` manager with its content pane. Figure 23-35 shows how instances of the `GreetingPanel`, `BagelPanel`, `ToppingPanel`, and `CoffeePanel` classes are placed in the content pane.

**Figure 23-35** Placement of the custom panels


Welcome to Brandi's Bagel House
Bagel
● White
○ Wheat
Toppings
☐ Cream cheese
☐ Butter
☐ Peach jelly
☐ Blueberry jam
Coffee
○ None
● Regular coffee
○ Decaf coffee
○ Cappuccino

An instance of the `GreetingPanel` class is added to the north region.

An instance of the `BagelPanel` class is added to the west region.

An instance of the `ToppingPanel` class is added to the center region.

An instance of the `CoffeePanel` class is added to the east region.

We have not created a custom panel class to hold the Calculate and Exit buttons. The reason is that the Calculate button's event listener must call the `getBagelCost`, `getToppingCost`, and `getCoffeeCost` methods. In order to call those methods, the event listener must have access to the `BagelPanel`, `ToppingPanel`, and `CoffeePanel` objects that are created in the `OrderCalculatorGUI` class. The approach taken in this example is to have the `OrderCalculatorGUI` class itself create the buttons. The code for the `OrderCalculatorGUI` class is shown in Code Listing 23-20.

**Code Listing 23-20** (`OrderCalculatorGUI.java`)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   The OrderCalculatorGUI class creates the GUI for the
   Brandi's Bagel House application.
*/

public class OrderCalculatorGUI extends JFrame
{
   private BagelPanel bagels;     // Bagel panel
   private ToppingPanel toppings; // Topping panel
   private CoffeePanel coffee;    // Coffee panel
   private GreetingPanel banner;  // To display a greeting
   private JPanel buttonPanel;    // To hold the buttons
   private JButton calcButton;    // To calculate the cost
   private JButton exitButton;    // To exit the application
   private final double TAX_RATE = 0.06; // Sales tax rate

   /**
      Constructor
   */

   public OrderCalculatorGUI()
   {
      // Display a title.
      setTitle("Order Calculator");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Create the custom panels.
      banner = new GreetingPanel();
      bagels = new BagelPanel();
      toppings = new ToppingPanel();
      coffee = new CoffeePanel();

      // Create the button panel.
      buildButtonPanel();

      // Add the components to the content pane.
      add(banner, BorderLayout.NORTH);
      add(bagels, BorderLayout.WEST);
      add(toppings, BorderLayout.CENTER);
      add(coffee, BorderLayout.EAST);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack the contents of the window and display it.
      pack();
      setVisible(true);
   }

   /**
      The buildButtonPanel method builds the button panel.
   */

   private void buildButtonPanel()
   {
      // Create a panel for the buttons.
      buttonPanel = new JPanel();

      // Create the buttons.
      calcButton = new JButton("Calculate");
      exitButton = new JButton("Exit");

      // Register the action listeners.
      calcButton.addActionListener(new CalcButtonListener());
      exitButton.addActionListener(new ExitButtonListener());

      // Add the buttons to the button panel.
      buttonPanel.add(calcButton);
      buttonPanel.add(exitButton);
   }

   /**
      Private inner class that handles the event when
      the user clicks the Calculate button.
   */

   private class CalcButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Variables to hold the subtotal, tax, and total
         double subtotal, tax, total;

         // Calculate the subtotal.
         subtotal = bagels.getBagelCost() +
                    toppings.getToppingCost() +
                    coffee.getCoffeeCost();

         // Calculate the sales tax.
         tax = subtotal * TAX_RATE;

         // Calculate the total.
         total = subtotal + tax;

         // Display the charges.
         JOptionPane.showMessageDialog(null,
            String.format("Subtotal: $%,.2f\n" +
                          "Tax: $%,.2f\n" +
                          "Total: $%,.2f",
                          subtotal, tax, total));
      }
   }

   /**
      Private inner class that handles the event when
      the user clicks the Exit button.
   */

   private class ExitButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0);
      }
   }

   /**
      main method
   */

   public static void main(String[] args)
   {
      new OrderCalculatorGUI();
   }
}
```

When the application runs, the window shown in Figure 23-36 appears. Figure 23-37 shows the `JOptionPane` dialog box that is displayed when the user selects a wheat bagel with butter, cream cheese, and decaf coffee.

**Figure 23-36** The Order Calculator window (Oracle Corporate Counsel)

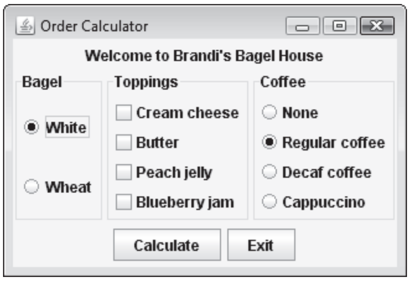
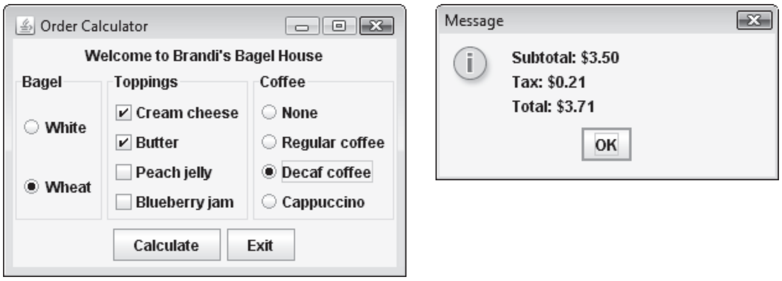

Order Calculator
Welcome to Brandi's Bagel House
Bagel
● White
○ Wheat
Toppings
☐ Cream cheese
☐ Butter
☐ Peach jelly
☐ Blueberry jam
Coffee
○ None
● Regular coffee
○ Decaf coffee
○ Cappuccino
Calculate
Exit

**Figure 23-37** The subtotal, tax, and total displayed (Oracle Corporate Counsel)


Order Calculator
Welcome to Brandi's Bagel House
Bagel
○ White
● Wheat
Toppings
☑ Cream cheese
☑ Butter
☐ Peach jelly
☐ Blueberry jam
Coffee
○ None
○ Regular coffee
● Decaf coffee
○ Cappuccino
Calculate
Exit
Message
OK

Subtotal: $3.50
Tax: $0.21
Total: $3.71

## 23.7 Splash Screens

**CONCEPT: A splash screen is a graphic image that is displayed while an application loads into memory and starts up.**

Most major applications display a splash screen, which is a graphic image that is displayed while the application is loading into memory. Splash screens usually show company logos and keep the user's attention while the application starts up. Splash screens are particularly important for large applications that take a long time to load, because they assure the user that the program is not malfunctioning.

Beginning with Java 6, you can display splash screens with your Java applications. First, you have to use a graphics program to create the image that you want to display. Java supports splash screens in the GIF, PNG, or JPEG formats. (If you are using Windows, you can create images with Microsoft Paint, which supports all of these formats.)

To display the splash screen you use the `java` command in the following way when you run the application:

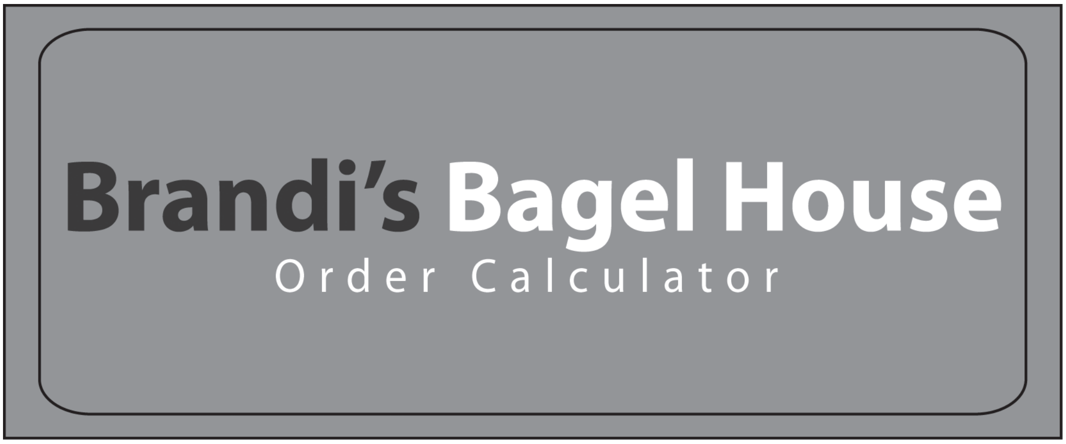
```
java -splash:GraphicFileName ClassFileName
```

*GraphicFileName* is the name of the file that contains the graphic image, and ***ClassFileName*** is the name of the *.class* file that you are running. For example, in the same source code folder as the *Brandi's Bagel House* application, you will find a file named *BrandiLogo.jpg*. This image, which is shown in Figure 23-38, is a logo for the *Brandi's Bagel House* application. To display the splash screen when the application starts, you would use the following command:

```
java splash:BrandiLogo.jpg Bagel
```

When you run this command, the graphic file will immediately be displayed in the center of the screen. It will remain displayed until the application's window appears.

**Figure 23-38** Splash screen for the *Brandi's Bagel House* application (Oracle Corporate Counsel)



## 23.8 Using Console Output to Debug a GUI Application

**CONCEPT: When debugging a GUI application, you can use `System.out.println` to send diagnostic messages to the console.**

When an application is not performing correctly, programmers sometimes write statements that display *diagnostic messages* into the application. For example, if an application is not giving the correct result for a calculation, diagnostic messages can be displayed at various points in the program's execution showing the values of all the variables used in the calculation. If the trouble is caused by a variable that has not been properly initialized, or that has not been assigned the correct value, the diagnostic messages reveal this problem. This helps the programmer see what is going on "under the hood" while an application is running.

The `System.out.println` method can be a valuable tool for displaying diagnostic messages in a GUI application. Because the `System.out.println` method sends its output to the console, diagnostic messages can be displayed without interfering with the application's GUI windows.

Code Listing 23-21 shows an example. This is a modified version of the `KiloConverter` class, discussed earlier in this chapter. Inside the `actionPerformed` method, which is in the `CalcButtonListener` inner class, calls to the `System.out.println` method have been written. The new code, which appears in lines 99 through 104 and 113 through 115, is shown in bold. These new statements display the value that the application has retrieved from the text field, and is working within its calculation. (This file is stored in the source code folder *Chapter 23\KiloConverter Phase 3*.)

**Code Listing 23-21** `(KiloConverter.java)`

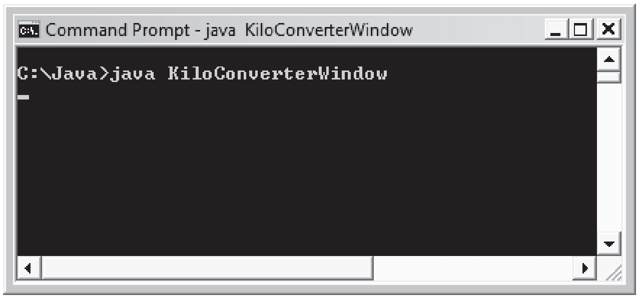
```
 1 import javax.swing.*;     // Needed for Swing classes
 2 import java.awt.event.*;  // Needed for ActionListener Interface
 3 
 4 /**
 5    The KiloConverter class displays a JFrame that
 6    lets the user enter a distance in kilometers. When
 7    the Calculate button is clicked, a dialog box is
 8    displayed with the distance converted to miles.
 9 */
10 
11 public class KiloConverter extends JFrame
12 {
13    private JPanel panel;             // To reference a panel
14    private JLabel messageLabel;      // To reference a label
15    private JTextField kiloTextField; // To reference a text field
16    private JButton calcButton;       // To reference a button
17    private final int WINDOW_WIDTH = 310;  // Window width
18    private final int WINDOW_HEIGHT = 100; // Window height
19 
20    /**
21       Constructor
22    */
23 
24    public KiloConverter()
25    {
26       // Set the window title.
27       setTitle("Kilometer Converter");
28 
29       // Set the size of the window.
30       setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
31 
32       // Specify what happens when the close button is clicked.
33       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```

      // Build the panel and add it to the frame.
      buildPanel();

      // Add the panel to the frame's content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      The buildPanel method adds a label, a text field,
      and a button to a panel.
   */

   private void buildPanel()
   {
      // Create a label to display instructions.
      messageLabel = new JLabel("Enter a distance " +
                                "in kilometers");

      // Create a text field 10 characters wide.
      kiloTextField = new JTextField(10);

      // Create a button with the caption "Calculate".
      calcButton = new JButton("Calculate");

      // Add an action listener to the button.
      calcButton.addActionListener(new CalcButtonListener());

      // Create a JPanel object and let the panel
      // field reference it.
      panel = new JPanel();

      // Add the label, text field, and button
      // components to the panel.
      panel.add(messageLabel);
      panel.add(kiloTextField);
      panel.add(calcButton);
   }

   /**
      CalcButtonListener is an action listener class for
      the Calculate button.
   */

   private class CalcButtonListener implements ActionListener
   {
      /**
         The actionPerformed method executes when the user
         clicks on the Calculate button.
         @param e The event object.
      */

      public void actionPerformed(ActionEvent e)
      {
         final double CONVERSION = 0.6214;
         String input;  // To hold the user's input
         double miles;  // The number of miles

         // Get the text entered by the user into the
         // text field.
         input = kiloTextField.getText();

         // For debugging, display the text entered, and
         // its value converted to a double.
         System.out.println("Reading " + input +
                            " from the text field.");
         System.out.println("Converted value: " +
                            Double.parseDouble(input));

         // Convert the input to miles.
         miles = Double.parseDouble(input) * CONVERSION;

         // Display the result.
         JOptionPane.showMessageDialog(null, input +
                 " kilometers is " + miles + " miles.");

         // For debugging, display a message indicating
         // the application is ready for more input.
         System.out.println("Ready for the next input.");
        }
      } // End of CalcButtonListener class

   /**
      The main method creates an instance of the
      KiloConverter class, which displays
      its window on the screen.
   */

   public static void main(String[] args)
   {
      new KiloConverter();
   }
}
```

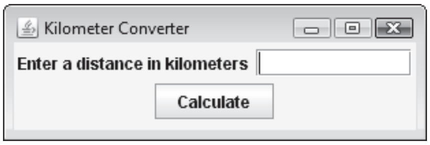
Let's take a closer look. In lines 101 and 102, a message is displayed to the console showing the value that was read from the text field. In lines 103 and 104, another message is displayed showing the value after it is converted to a `double`. Then, in line 115, a message is displayed indicating that the application is ready for its next input. Figure 23-39 shows an example session with the application on a computer running Microsoft Windows. Both the console window and the application windows are shown.

**Figure 23-39** Messages displayed to the console during the application's execution
(Oracle Corporate Counsel)

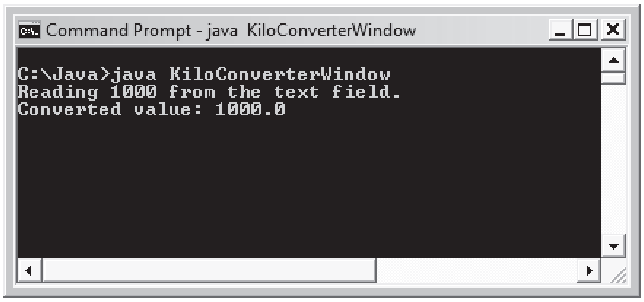
1. A command is typed in the console window to execute the application. The application's window appears.


Command Prompt - java KiloConverterWindow

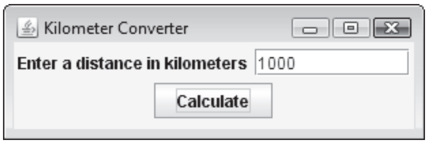
```
C:\Java>java KiloConverterWindow
_
```


Kilometer Converter
Enter a distance in kilometers
Calculate

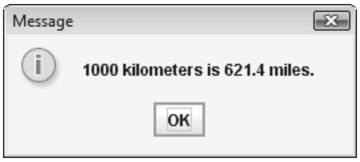
2. The user types a value into the text field and clicks the Calculate button. Debugging messages appear in the console window, and a message dialog appears showing the value converted to miles.


Command Prompt - java KiloConverterWindow

```
C:\Java>java KiloConverterWindow
Reading 1000 from the text field.
Converted value: 1000.0
```

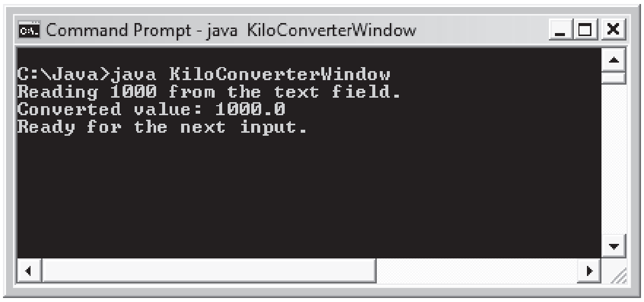

Kilometer Converter
Calculate

Enter a distance in kilometers 1000

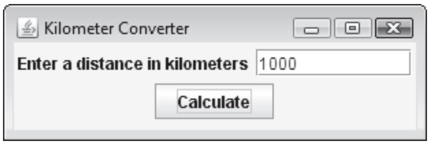


1000 kilometers is 621.4 miles.

3. The user dismisses the dialog box and a message is displayed in the console window indicating that the application is ready for the next input.


Command Prompt - java KiloConverterWindow

```
C:\Java>java KiloConverterWindow
Reading 1000 from the text field.
Converted value: 1000.0
Ready for the next input.
```


Kilometer Converter
Calculate

Enter a distance in kilometers 1000

The messages that are displayed to the console are meant for only the programmer to see, while he or she is debugging the application. Once the programmer is satisfied that the application is running correctly, the calls to `System.out.println` can be taken out.

## 23.9 Common Errors to Avoid

- **Misspelling `javax.swing` in an `import` statement.** Don't forget the letter `x` that appears after `java` in this import statement.
- **Forgetting to specify the action taken when the user clicks on a `JFrame`'s close button.** By default, a window is hidden from view when the close button is clicked, but the application is not terminated. If you wish to exit the application when a `JFrame's` close button is clicked, you must call the `setDefaultCloseOperation` method and pass `JFrame.EXIT_ON_CLOSE` as the argument.
- **Forgetting to write an event listener for each event you wish an application to respond to.** In order to respond to an event, you must write an event listener that implements the proper type of interface, registered to the component that generates the event.
- **Forgetting to register an event listener.** Even if you write an event listener, it will not execute unless it has been registered with the correct component.
- **When writing an event listener method that is required by an interface, not using the method header specified by the interface.** The header of an `actionPerformed` method must match that specified by the `ActionListener` interface. Also, the header of an `itemStateChanged` method must match that specified by the `ItemListener` method.
- **Placing components directly into the regions of a container governed by a `BorderLayout` manager when you do not want the components resized or you want to add more than one component per region.** If you do not want the components that you place in a `BorderLayout` region to be resized, place them in a `JPanel` component and then add the `JPanel` component to the region.
- **Placing components directly into the cells of a container governed by a `GridLayout` manager when you do not want the components resized or you want to add more than one component per cell.** If you do not want the components that you place in a `GridLayout` cell to be resized, place them in a `JPanel` component, and then add the `JPanel` component to the cell.
- **Forgetting to add `JRadioButton` components to a `ButtonGroup` object.** A mutually exclusive relationship is created between radio buttons only when they are added to a `ButtonGroup` object.

## Review Questions and Exercises

### Multiple Choice and True/False

1. With Swing, you use this class to create a frame.
   a. `Frame`
   b. `SwingFrame`
   c. `JFrame`
   d. `JavaFrame`
2. This is the part of a `JFrame` object that holds the components that have been added to the `JFrame` object.
   a. content pane
   b. viewing area
   c. component array
   d. object collection

3. This is a `JPanel` object's default layout manager.
   a. `BorderLayout`
   b. `GridLayout`
   c. `FlowLayout`
   d. None
4. This is the default layout manager for a `JFrame` object's content pane.
   a. `BorderLayout`
   b. `GridLayout`
   c. `FlowLayout`
   d. None
5. If a container is governed by a `BorderLayout` manager and you add a component to it, but you do not pass the second argument specifying the region, this is the region in which the component will be added.
   a. north
   b. south
   c. east
   d. center
6. Components in this/these regions of a `BorderLayout` manager are resized horizontally so they fill up the entire region.
   a. north and south
   b. east and west
   c. center only
   d. north, south, east, and west
7. Components in this/these regions of a `BorderLayout` manager are resized vertically so they fill up the entire region.
   a. north and south
   b. east and west
   c. center only
   d. north, south, east, and west
8. Components in this/these regions of a `BorderLayout` manager are resized both horizontally and vertically so they fill up the entire region.
   a. north and south
   b. east and west
   c. center only
   d. north, south, east, and west
9. This is the default alignment of a `FlowLayout` manager.
   a. left
   b. center
   c. right
   d. no alignment
10. Adding radio button components to this type of object creates a mutually exclusive relationship between them.
    a. `MutualExclude`
    b. `RadioGroup`
    c. `LogicalGroup`
    d. `ButtonGroup`

11. You use this class to create `Border` objects.
    a. `BorderFactory`
    b. `BorderMaker`
    c. `BorderCreator`
    d. `BorderSource`
12. **True or False:** A panel cannot be displayed by itself.
13. **True or False:** You can place multiple components inside a `GridLayout` cell.
14. **True or False:** You can place multiple components inside a `BorderLayout` region.
15. **True or False:** You can place multiple components inside a container governed by a `FlowLayout` manager.
16. **True or False:** You can place a panel inside a region governed by a `BorderLayout` manager.
17. **True or False:** A component placed in a `GridLayout` manager's cell will not be resized to fill up any extra space in the cell.
18. **True or False:** You normally add `JCheckBox` components to a `ButtonGroup` object.
19. **True or False:** A mutually exclusive relationship is automatically created among all JRadioButton components in the same container.
20. **True or False:** You can write a class that extends the `JPanel` class.

## Find the Error

1. The following statement is in a class that uses Swing components:

```
import java.swing.*;
```

2. The following is an inner class that will be registered as an action listener for a `JButton` component:

```
private class ButtonListener implements ActionListener
{
   public void actionPerformed()
   {
      // Code appears here.
   }
}
```

3. The intention of the following statement is to give the `panel` object a `GridLayout` manager with 10 columns and 5 rows:

```
panel.setLayout(new GridLayout(10, 5));
```

4. The `panel` variable references a `JPanel` governed by a `BorderLayout` manager. The following statement attempts to add the button component to the north region of panel:

```
panel.add(button, NORTH);
```

5. The `panel` variable references a `JPanel` object. The intention of the following statement is to create a titled border around `panel`:

```
panel.setBorder(new BorderFactory("Choices"));
```

### Algorithm Workbench

1. The variable `myWindow` references a `JFrame` object. Write a statement that sets the size of the object to 500 pixels wide and 250 pixels high.
2. The variable `myWindow` references a `JFrame` object. Write a statement that causes the application to end when the user clicks on the `JFrame` object's close button.
3. The variable `myWindow` references a `JFrame` object. Write a statement that displays the object's window on the screen.
4. The variable `myButton` references a `JButton` object. Write the code to set the object's background color to white and foreground color to red.
5. Assume that a class inherits from the `JFrame` class. Write code that can appear in the class constructor, which gives the content pane a `FlowLayout` manager. Components added to the content pane should be aligned with the left edge of each row.
6. Assume that a class inherits from the `JFrame` class. Write code that can appear in the class constructor, which gives the content pane a `GridLayout` manager with five rows and 10 columns.
7. Assume that the variable `panel` references a `JPanel` object that uses a `BorderLayout` manager. In addition, the variable `button` references a `JButton` object. Write code that adds the `button` object to the `panel` object's west region.
8. Write code that creates three radio buttons with the text "Option 1", "Option 2", and "Option 3". The radio button that displays the text "Option 1" should be initially selected. Make sure these components are grouped so that a mutually exclusive relationship exists among them.
9. Assume that panel references a `JPanel` object. Write code that creates a two pixel thick blue line border around it.

### Short Answer

1. If you do not change the default close operation, what happens when the user clicks on the close button on a `JFrame` object?
2. Why is it sometimes necessary to place a component inside a panel and then place the panel inside a container governed by a `BorderLayout` manager?
3. In what type of situation would you present a group of items to the user with radio buttons? With check boxes?
4. How can you create a specialized panel component that can be used to hold other components and their related code?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Retail Price Calculator

Create a GUI application where the user enters the wholesale cost of an item and its markup percentage into text fields. (For example, if an item's wholesale cost is $5 and its markup percentage is 100 percent, then its retail price is $10.) The application should have a button that displays the item's retail price when clicked.

### 2. Monthly Sales Tax

VideoNote
The Monthly Sales Tax Problem

A retail company must file a monthly sales tax report listing the total sales for the month, and the amount of state and county sales tax collected. The state sales tax rate is 4 percent and the county sales tax rate is 2 percent. Create a GUI application that allows the user to enter the total sales for the month into a text field. From this figure, the application should calculate and display the following:

- The amount of county sales tax
- The amount of state sales tax
- The total sales tax (county plus state)

In the application's code, represent the county tax rate (0.02) and the state tax rate (0.04) as named constants.

### 3. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. If an acre of land is valued at $10,000, its assessment value is $6,000. The property tax is then $0.64 for each $100 of the assessment value. The tax for the acre assessed at $6,000 will be $38.40. Create a GUI application that displays the assessment value and property tax when a user enters the actual value of a property.

### 4. Travel Expenses

Create a GUI application that calculates and displays the total travel expenses of a business person on a trip. Here is the information that the user must provide:

- Number of days on the trip
- Amount of airfare, if any
- Amount of car rental fees, if any
- Number of miles driven, if a private vehicle was used
- Amount of parking fees, if any
- Amount of taxi charges, if any
- Conference or seminar registration fees, if any
- Lodging charges, per night

The company reimburses travel expenses according to the following policy:

- $37 per day for meals
- Parking fees, up to $10.00 per day
- Taxi charges up to $20.00 per day
- Lodging charges up to $95.00 per day
- If a private vehicle is used, $0.27 per mile driven

The application should calculate and display the following:

- Total expenses incurred by the business person
- The total allowable expenses for the trip
- The excess that must be paid by the business person, if any
- The amount saved by the business person if the expenses are under the total allowed

### 5. Theater Revenue

A movie theater only keeps a percentage of the revenue earned from ticket sales. The remainder goes to the movie company. Create a GUI application that allows the user to enter the following data into text fields:

- Price per adult ticket
- Number of adult tickets sold
- Price per child ticket
- Number of child tickets sold

The application should calculate and display the following data for one night's box office business at a theater:

- **Gross revenue for adult tickets sold.** This is the amount of money taken in for all adult tickets sold.
- **Net revenue for adult tickets sold.** This is the amount of money from adult ticket sales left over after the payment to the movie company has been deducted.
- **Gross revenue for child tickets sold.** This is the amount of money taken in for all child tickets sold.
- **Net revenue for child tickets sold.** This is the amount of money from child ticket sales left over after the payment to the movie company has been deducted.
- **Total gross revenue.** This is the sum of gross revenue for adult and child tickets sold.
- **Total net revenue.** This is the sum of net revenue for adult and child tickets sold.

Assume the theater keeps 20 percent of its box office receipts. Use a constant in your code to represent this percentage.

### 6. Joe's Automotive

Joe's Automotive performs the following routine maintenance services:

- Oil change—$26.00
- Lube job—$18.00
- Radiator flush—$30.00
- Transmission flush—$80.00
- Inspection—$15.00
- Muffler replacement—$100.00
- Tire rotation—$20.00

Joe also performs other nonroutine services and charges for parts and for labor ($20 per hour). Create a GUI application that displays the total for a customer's visit to Joe's.

### 7. Long Distance Calls

A long-distance provider charges the following rates for telephone calls:

| Rate Category | Rate per Minute |
|---|---|
| Daytime (6:00 A.M. through 5:59 P.M.) | $0.07 |
| Evening (6:00 P.M. through 11:59 P.M.) | $0.12 |
| Off-Peak (12:00 A.M. through 5:59 A.M.) | $0.05 |

Create a GUI application that allows the user to select a rate category (from a set of radio buttons), and enter the number of minutes of the call into a text field. A dialog box should display the charge for the call.

### 8. Latin Translator

Look at the following list of Latin words and their meanings.

| Latin | English |
|---|---|
| sinister | left |
| dexter | right |
| medium | center |

Write a GUI application that translates the Latin words to English. The window should have three buttons, one for each Latin word. When the user clicks a button, the program displays the English translation in a label.

### 9. MPG Calculator

Write a GUI application that calculates a car's gas mileage. The application should let the user enter the number of gallons of gas the car holds, and the number of miles it can be driven on a full tank. When a *Calculate MPG* button is clicked, the application should display the number of miles that the car may be driven per gallon of gas. Use the following formula to calculate MPG:

$$MPG = \frac{Miles}{Gallons}$$

### 10. Celsius to Fahrenheit

Write a GUI application that converts Celsius temperatures to Fahrenheit temperatures. The user should be able to enter a Celsius temperature, click a button, and then see the equivalent Fahrenheit temperature. Use the following formula to make the conversion:

$$F = \frac{9}{5}C + 32$$

F is the Fahrenheit temperature and C is the Celsius temperature.

CHAPTER

# 24

# Advanced Swing GUI Applications

**TOPICS**

24.1 The Swing and AWT Class Hierarchy
24.2 Read-Only Text Fields
24.3 Lists
24.4 Combo Boxes
24.5 Displaying Images in Labels and Buttons
24.6 Mnemonics and Tool Tips
24.7 File Choosers and Color Choosers
24.8 Menus
24.9 More about Text Components: Text Areas and Fonts
24.10 Sliders
24.11 Look and Feel
24.12 Common Errors to Avoid

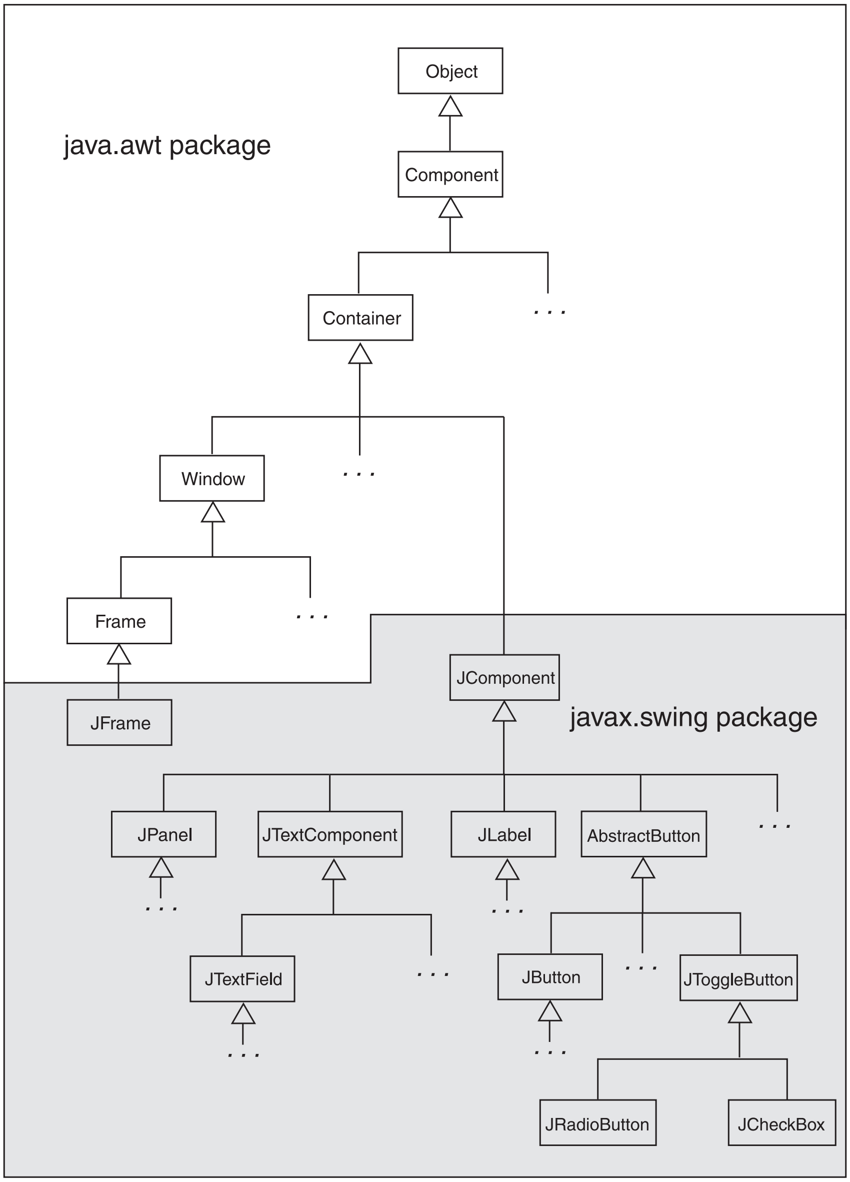
**NOTE:** This chapter discusses GUI development using the Swing classes. Oracle has announced that JavaFX is replacing Swing as the standard GUI library for Java. Swing will remain part of the Java API for the foreseeable future, however, so we are providing this chapter for you to use as you make the transition from Swing to JavaFX. To learn about JavaFX, see Chapters 12, 13, and 14.

## 24.1 The Swing and AWT Class Hierarchy

Now that you have used some of the fundamental GUI components, let's look at how they fit into the class hierarchy. Figure 24-1 shows the parts of the Swing and AWT class hierarchy that contain the `JFrame`, `JPanel`, `JLabel`, `JTextField`, `JButton`, `JRadioButton`, and `JCheckBox` classes. Because of the inheritance relationships that exist, there are many other classes in the figure as well.

The classes that are in the unshaded top part of the figure are AWT classes and are in the `java.awt` package. The classes that are in the shaded bottom part of the figure are Swing classes and are in the `javax.swing` package. Notice that all of the components we have dealt with ultimately inherit from the `Component` class.

**Figure 24-1** Part of the Swing and AWT class hierarchy (Oracle Corporate Counsel)


java.awt package
Object
Component
Container
. . .
Window
. . .
Frame
. . .
JFrame
JComponent
javax.swing package
JPanel
JTextComponent
JLabel
AbstractButton
. . .
. . .
JTextField
. . .
. . .
JButton
. . .
JToggleButton
. . .
. . .
JRadioButton
JCheckBox

## 24.2 Read-Only Text Fields

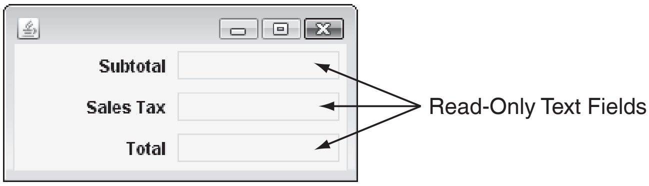
**CONCEPT:** **A read-only text field displays text that can be changed by code in the application, but cannot be edited by the user.**

A read-only text field is not a new component, but a different way to use the `JTextField` component. The `JTextField` component has a method named `setEditable`, which has the following general format:

```
setEditable(boolean editable)
```

You pass a `boolean` argument to this method. By default a text field is editable, which means that the user can enter data into it. If you call the `setEditable` method and pass `false` as the argument, then the text field becomes read-only. This means it is not editable by the user. Figure 24-2 shows a window that has three read-only text fields.

**Figure 24-2** A window with three read-only text fields (Oracle Corporate Counsel)



The following code could be used to create the read-only text fields shown in the figure:

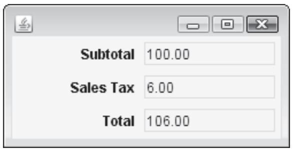
```
// Create a read-only text field for the subtotal.
JTextField subtotalField = new JTextField(10);
subtotalField.setEditable(false);

// Create a read-only text field for the sales tax.
JTextField taxField = new JTextField(10);
taxField.setEditable(false);

// Create a read-only text field for the total.
JTextField totalField = new JTextField(10);
totalField.setEditable(false);
```

A read-only text field looks like a label with a border drawn around it. You can use the `setText` method to display data inside it. Here is an example:

```
subtotalField.setText("100.00");
taxField.setText("6.00");
totalField.setText("106.00");
```

This code causes the text fields to appear as shown in Figure 24-3.

**Figure 24-3** Read-only text fields with data displayed (Oracle Corporate Counsel)



Subtotal 100.00
Sales Tax 6.00
Total 106.00

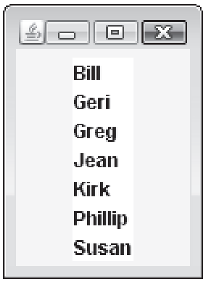
## 24.3 Lists

**CONCEPT: A list component displays a list of items and allows the user to select an item from the list.**

VideoNote
The `JList` Component

A list is a component that displays a list of items and also allows the user to select one or more items from the list. Java provides the `JList` component for creating lists. Figure 24-4 shows an example. The `JList` component in the figure shows a list of names. At runtime, the user may select an item in the list, which causes the item to appear highlighted. In the figure, the first name is selected.

**Figure 24-4** A `JList` component (Oracle Corporate Counsel)



When you create an instance of the `JList` class, you pass an array of objects to the constructor. Here is the general format of the constructor call:

```
JList (Object[] array)
```

The `JList` component uses the array to create the list of items. In this text we always pass an array of `String` objects to the `JList` constructor. For example, the list component shown in Figure 24-4 could be created with the following code:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

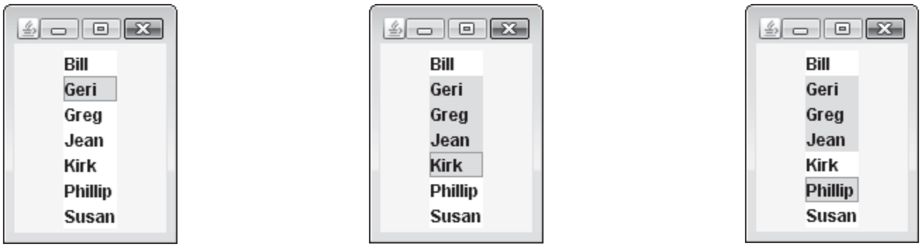
## Selection Modes

The `JList` component can operate in any of the following selection modes:

- Single Selection Mode. In this mode only one item can be selected at a time. When an item is selected, any other item that is currently selected is deselected.
- Single Interval Selection Mode. In this mode multiple items can be selected, but they must be in a single interval. An interval is a set of contiguous items.
- Multiple Interval Selection Mode. In this mode multiple items may be selected with no restrictions. This is the default selection mode.

Figure 24-5 shows an example of a list in each type of selection mode.

**Figure 24-5** Selection modes (Oracle Corporate Counsel)



Single selection mode allows only one item to be selected at a time.

Single interval selection mode allows a single interval of contiguous items to be selected.

Multiple interval selection mode allows multiple items to be selected with no restrictions.

The default mode is multiple interval selection. To keep our applications simple, we will use single selection mode for now. You change a `JList` component's selection mode with the `setSelectionMode` method. The method accepts an `int` argument that determines the selection mode.

The `ListSelectionModel` class, which is in the `javax.swing` package, provides the following constants that you can use as arguments to the `setSelectionMode` method:

- `ListSelectionModel.SINGLE_SELECTION`
- `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
- `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`

Assuming that `nameList` references a `JList` component, the following statement sets the component to single selection mode:

```
nameList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

## Responding to List Events

When an item in a `JList` object is selected it generates a list selection event. You handle list selection events with a list selection listener class, which must meet the following requirements:

- It must implement the `ListSelectionListener` interface.
- It must have a method named `valueChanged`. This method must take an argument of the `ListSelectionEvent` type.

**NOTE:** The `ListSelectionListener` interface is in the `javax.swing.event` package, so you must have an `import` statement for that package in your source code.

Once you have written a list selection listener class, you create an object of that class and then pass it as an argument to the `JList` component's `addListSelectionListener` method. When the `JList` component generates an event, it automatically executes the `valueChanged` method of the list selection listener object, passing the event object as an argument. You will see an example in a moment.

## Retrieving the Selected Item

You may use either the `getSelectedValue` method or the `getSelectedIndex` method to determine which item in a list is currently selected. The `getSelectedValue` method returns a reference to the item that is currently selected. For example, assume that `nameList` references the `JList` component shown earlier in Figure 24-4. The following code retrieves a reference to the name that is currently selected and assigns it to the `selectedName` variable:

```
String selectedName;
selectedName = (String) nameList.getSelectedValue();
```

Note that the return value of the `getSelectedValue` method is an `Object` reference. In this code we had to cast the return value to the `String` type in order to store it in the `selectedName` variable. If no item in the list is selected, the method returns `null`.

The `getSelectedIndex` method returns the index of the selected item, or −1 if no item is selected. Internally, the items that are stored in a list are numbered. Each item's number is called its index. The first item (which is the item stored at the top of the list) has the index 0, the second item has the index 1, and so forth. You can use the index of the selected item to retrieve the item from an array. For example, assume that the following code was used to build the `nameList` component shown in Figure 24-4:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

Because the `names` array holds the values displayed in the `namesList` component, the following code could be used to determine the selected item:

```
int index;
String selectedName;
index = nameList.getSelectedIndex();
if (index != –1)
   selectedName = names[index];
```

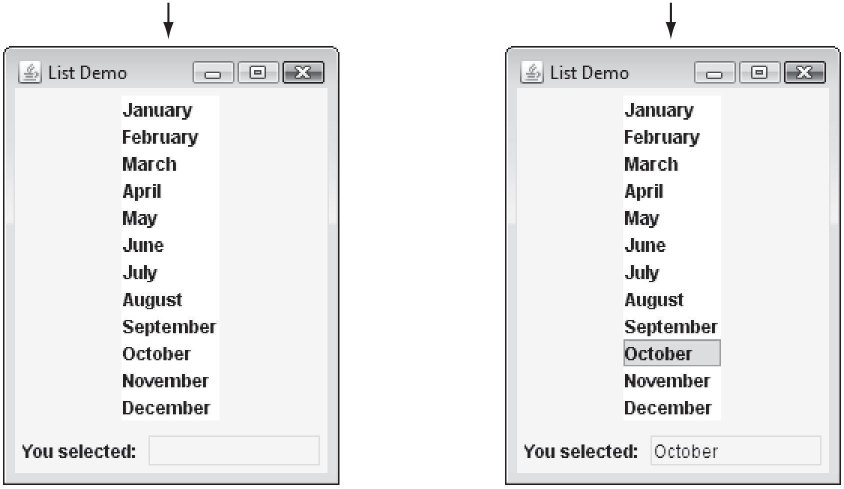
The `ListWindow` class shown in Code Listing 24-1 demonstrates the concepts we have discussed so far. It uses a `JList` component with a list selection listener. When an item is selected from the list, it is displayed in a read-only text field. The `main` method creates an instance of the `ListWindow` class, which displays the window shown on the left in Figure 24-6. After the user selects October from the list, the window appears as that shown on the right in the figure.

**Code Listing 24-1** **(`ListWindow.java`)**

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

/**
   This class demonstrates the List Component.
*/
```

```

public class ListWindow extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JList monthList;             // The months
   private JTextField selectedMonth;    // The selected month
   private JLabel label;                // A message

   // The following array holds the values that will
   // be displayed in the monthList list component.
   private String[] months = { "January", "February",
                            "March", "April", "May", "June", "July",
                            "August", "September", "October", "November",
                            "December" };

   /**
      Constructor
   */

   public ListWindow()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the month and selectedMonth panels.
      buildMonthPanel();
      buildSelectedMonthPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.CENTER);
      add(selectedMonthPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing
      the names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to single selection.
      monthList.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);

      // Register the list selection listener.
      monthList.addListSelectionListener(
                                  new ListListener());

      // Add the list to the panel.
      monthPanel.add(monthList);
   }

   /**
      The buildSelectedMonthPanel method adds an
      uneditable text field to a panel.
   */

   private void buildSelectedMonthPanel()
   {
      // Create a panel to hold the text field.
      selectedMonthPanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the text field.
      selectedMonth = new JTextField(10);

      // Make the text field uneditable.
      selectedMonth.setEditable(false);

      // Add the label and text field to the panel.
      selectedMonthPanel.add(label);
      selectedMonthPanel.add(selectedMonth);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the list.
   */

   private class ListListener
                    implements ListSelectionListener
   {
      public void valueChanged(ListSelectionEvent e)
      {
         // Get the selected month.
         String selection =
               (String) monthList.getSelectedValue();

         // Put the selected month in the text field.
         selectedMonth.setText(selection);
      }
   }

   /**
      The main method creates an instance of the
      ListWindow class which causes it to display
      its window.
   */

     public static void main(String[] args)
     {
          new ListWindow();
     }
}
```

**Figure 24-6** Window displayed by the `ListWindow` class (Oracle Corporate Counsel)



Window as initially displayed.

Window after the user selects October.

List Demo
January
February
March
April
May
June
July
August
September
October
November
December
List Demo
January
February
March
April
May
June
July
August
September
October
November
December

You selected:

You selected: October

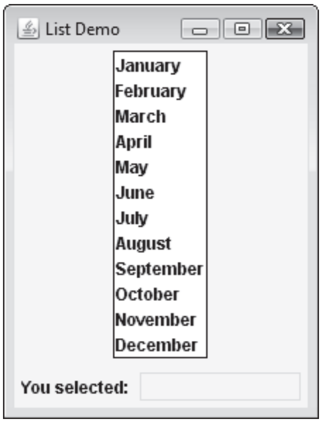
## Placing a Border around a List

As with other components, you can use the `setBorder` method, which was discussed in Chapter 23, to draw a border around a `JList`. For example the following statement can be used to draw a black 1-pixel thick line border around the `monthList` component:

```
monthList.setBorder(BorderFactory.createLineBorder(Color.BLACK, 1));
```

This code will cause the list to appear as shown in Figure 24-7.

**Figure 24-7** List with a line border (Oracle Corporate Counsel)



## Adding a Scroll Bar to a List

By default, a list component is large enough to display all of the items it contains. Sometimes a list component contains too many items to be displayed at once, however. Most GUI applications display a scroll bar on list components that contain a large number of items. The user simply uses the scroll bar to scroll through the list of items.

List components do not automatically display a scroll bar. To display a scroll bar on a list component, you must follow the following general steps:

1. Set the number of visible rows for the list component.
2. Create a scroll pane object and add the list component to it.
3. Add the scroll pane object to any other containers, such as panels.

Let's take a closer look at how these steps can be used to apply a scroll bar to the list component created in the following code:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

First, we establish the size of the list component with the `JList` class's `setVisibleRowCount` method. The following statement sets the number of visible rows in the `nameList` component to three:

```
nameList.setVisibleRowCount(3);
```

This statement causes the `nameList` component to display only three items at a time.

Next, we create a scroll pane object and add the list component to it. A scroll pane object is a container that displays scroll bars on any component it contains. In Java we use the `JScrollPane` class to create a scroll pane object. We pass the object that we wish to add to the scroll pane as an argument to the `JScrollPane` constructor. The following statement demonstrates:

```
JScrollPane scrollPane = new JScrollPane(nameList);
```

This statement creates a `JScrollPane` object and adds the `nameList` component to it.

Next, we add the scroll pane object to any other containers that are necessary for our GUI. For example, the following code adds the scroll pane to a `JPanel`, which is then added to the `JFrame` object's content pane:

```
// Create a panel and add the scroll pane to it.
JPanel panel = new JPanel();
panel.add(scrollPane);

// Add the panel to this JFrame object's contentPane.
add(panel);
```

When the list component is displayed, it will appear as shown in Figure 24-8.

Although the list component displays only three items at a time, the user can scroll through all of the items it contains.

The `ListWindowWithScroll` class shown in Code Listing 24-2 is a modification of the `ListWindow` class. In this class, the `monthList` component shows only six items at a time, but displays a scroll bar. The code shown in bold is the new lines that are used to add the scroll bar to the list. The `main` method creates an instance of the class, which displays the window shown in Figure 24-9.

**Figure 24-8** List component with a scroll bar (Oracle Corporate Counsel)

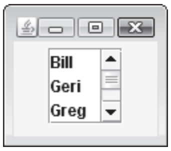
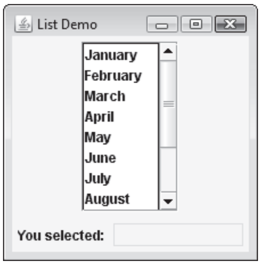


**Figure 24-9** List component with scroll bars (Oracle Corporate Counsel)



**Code Listing 24-2** (ListWindowWithScroll.java)

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

/**
   This class demonstrates the List Component.
*/

public class ListWindowWithScroll extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JList monthList;             // The months
   private JScrollPane scrollPane;      // A scroll pane
   private JTextField selectedMonth;    // The selected month
   private JLabel label;                // A message

   // The following array holds the values that will
   // be displayed in the monthList list component.
   private String[] months = { "January", "February",
             "March", "April", "May", "June", "July",
             "August", "September", "October", "November",
             "December" };

   /**
      Constructor
   */

   public ListWindowWithScroll()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the month and selectedMonth panels.
      buildMonthPanel();
      buildSelectedMonthPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.CENTER);
      add(selectedMonthPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing
      the names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to single selection.
      monthList.setSelectionMode(
                  ListSelectionModel.SINGLE_SELECTION);

      // Register the list selection listener.
      monthList.addListSelectionListener(
                                 new ListListener());

      // Set the number of visible rows to 6.
      monthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane = new JScrollPane(monthList);

      // Add the scroll pane to the panel.
      monthPanel.add(scrollPane);
   }

   /**
      The buildSelectedMonthPanel method adds an
      uneditable text field to a panel.
   */

   private void buildSelectedMonthPanel()
   {
      // Create a panel to hold the text field.
      selectedMonthPanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the text field.
      selectedMonth = new JTextField(10);

      // Make the text field uneditable.
      selectedMonth.setEditable(false);

      // Add the label and text field to the panel.
      selectedMonthPanel.add(label);
      selectedMonthPanel.add(selectedMonth);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the list.
   */

   private class ListListener
                      implements ListSelectionListener
   {
      public void valueChanged(ListSelectionEvent e)
      {
         // Get the selected month.
         String selection =
                (String) monthList.getSelectedValue();

         // Put the selected month in the text field.
         selectedMonth.setText(selection);
      }
   }

   /**
      The main method creates an instance of the
      ListWindowWithScroll class which causes it
      to display its window.
   */

     public static void main(String[] args)
     {
        new ListWindowWithScroll();
     }
}
```

**NOTE:** By default, when a `JList` component is added to a `JScrollPane` object, the scroll bar is only displayed when there are more items in the list than there are visible rows.

**NOTE:** When a `JList` component is added to a `JScrollPane` object, a border will automatically appear around the list.

## Adding Items to an Existing `JList` Component

The `JList` class's `setListData` method allows you to store items in an existing `JList` component. Here is the method's general format:

```
void setListData(Object[] data)
```

The argument passed into `data` is an array of objects that will become the items displayed in the `JList` component. Any items that are currently displayed in the component will be replaced by the new items.

In addition to replacing the existing items in a list, you can use this method to add items to an empty list. You can create an empty list by passing no argument to the `JList` constructor. Here is an example:

```
JList nameList = new JList();
```

This statement creates an empty `JList` component referenced by the `nameList` variable. You can then add items to the list, as shown here:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
nameList.setListData(names);
```

## Multiple Selection Lists

For simplicity, the previous examples used a `JList` component in single selection mode. Recall that the other two selection modes are single interval and multiple interval. Both of these modes allow the user to select multiple items. Let's take a closer look at each of these modes.

### Single Interval Selection Mode

You put a `JList` component in single interval selection mode by passing the constant `ListSelectionModel.SINGLE_INTERVAL_SELECTION` to the component's `setSelectionMode` method. In single interval selection mode, single or multiple items can be selected. An interval is a set of contiguous items. (See Figure 24-5 to see an example of an interval.)

To select an interval of items, the user selects the first item in the interval by clicking on it, and then selects the last item in the interval by holding down the Shift key while clicking on it. All of the items that appear in the list from the first item through the last item are selected.

In single interval selection mode, the `getSelectedValue` method returns the first item in the selected interval. The `getSelectedIndex` method returns the index of the first item in the selected interval. To get the entire selected interval, use the `getSelectedValues` method. This method returns an array of objects. The array will hold the items in the selected interval. You can also use the `getSelectedIndices` method, which returns an array of `int` values. The values in the array will be the indices of all the selected items in the list.

### Multiple Interval Selection Mode

You put a `JList` component in multiple interval selection mode by passing the constant `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION` to the component's `setSelectionMode` method. In multiple interval selection mode, multiple items can be selected and the items do not have to be in the same interval. (See Figure 24-5 for an example.)

In multiple interval selection mode, the user can select single items or intervals. When the user holds down the Ctrl key while clicking on an item, it selects the item without deselecting any items that are currently selected. This allows the user to select multiple items that are not in an interval.

In multiple interval selection mode, the `getSelectedValue` method returns the first selected item. The `getSelectedIndex` method returns the index of the first selected item. The `getSelectedValues` method returns an array of objects containing the items that are selected. The `getSelectedIndices` method returns an `int` array containing the indices of all the selected items in the list.

The `MultipleIntervalSelection` class, shown in Code Listing 24-3, demonstrates a `JList` component used in multiple interval selection mode. The `main` method creates an instance of the class that displays the window shown on the left in Figure 24-10. When the user selects items from the top `JList` component and then clicks the Get Selections button, the selected items appear in the bottom `JList` component.

**Code Listing 24-3** `(MultipleIntervalSelection.java)`

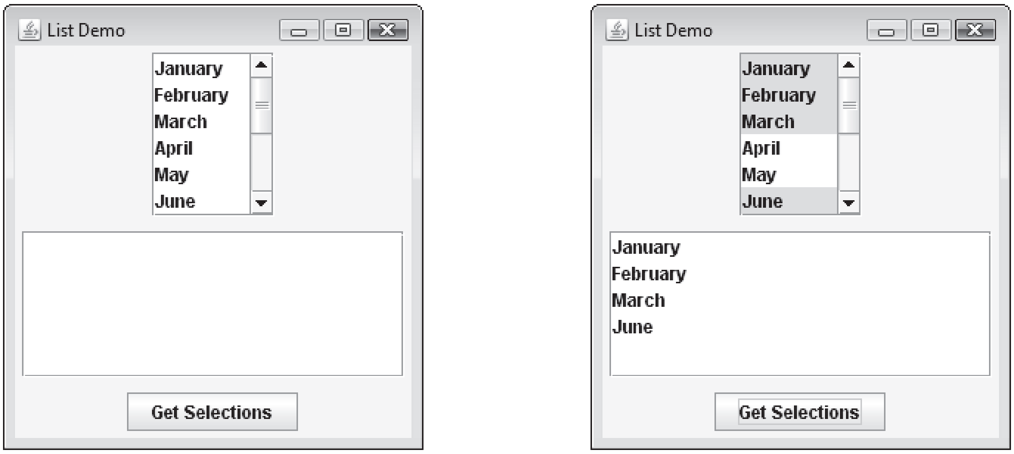
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   This class demonstrates the List Component in
   multiple interval selection mode.
*/

public class MultipleIntervalSelection extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JPanel buttonPanel;          // To hold the button

   private JList monthList;            // To hold months
   private JList selectedMonthList;    // Selected months

   private JScrollPane scrollPane1;    // Scroll pane - first list
   private JScrollPane scrollPane2;    // Scroll pane - second list

   private JButton button;             // A button

   // The following array holds the values that
   // will be displayed in the monthList list component.
   private String[] months = { "January", "February",
             "March", "April", "May", "June", "July",
             "August", "September", "October", "November",
             "December" };

   /**
      Constructor
   */

   public MultipleIntervalSelection()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildMonthPanel();
      buildSelectedMonthsPanel();
      buildButtonPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.NORTH);
      add(selectedMonthPanel,BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing the
      names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to multiple
      // interval selection.
      monthList.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);

      // Set the number of visible rows to 6.
      monthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane1 = new JScrollPane(monthList);

      // Add the scroll pane to the panel.
      monthPanel.add(scrollPane1);
   }

   /**
      The buildSelectedMonthsPanel method adds a list
      to a panel. This will hold the selected months.
   */

   private void buildSelectedMonthsPanel()
   {
      // Create a panel to hold the list.
      selectedMonthPanel = new JPanel();

      // Create the list.
      selectedMonthList = new JList();

      // Set the number of visible rows to 6.
      selectedMonthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane2 =
               new JScrollPane(selectedMonthList);

      // Add the scroll pane to the panel.
      selectedMonthPanel.add(scrollPane2);
   }

   /**
      The buildButtonPanel method adds a
      button to a panel.
   */

   private void buildButtonPanel()
   {
      // Create a panel to hold the list.
      buttonPanel = new JPanel();

      // Create the button.
      button = new JButton("Get Selections");

      // Add an action listener to the button.
      button.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(button);
   }

   /**
      Private inner class that handles the event when
      the user clicks the button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Get the selected values.
         Object[] selections =
                         monthList.getSelectedValues();

         // Store the selected items in selectedMonthList.
         selectedMonthList.setListData(selections);
      }
   }

   /**
      The main method creates an instance of the
      MultipleIntervalSelection class which causes it
      to display its window.
   */

   public static void main(String[] args)
```

```
    {
        new MultipleIntervalSelection();
    }
}
```

**Figure 24-10** The window displayed by the `MultipleIntervalSelection` class (Oracle Corporate Counsel)

This is the window as it is intially displayed.

This is the window after the user has selected some items from the top list and clicked the Get Selections button.



```
January
February
March
June
```

## 24.4 Combo Boxes

**CONCEPT:** **A combo box allows the user to select an item from a drop-down list.**

VideoNote
The JComboBox Component

A combo box presents a list of items that the user may select from. Unlike a list component, a combo box presents its items in a drop-down list. You use the `JComboBox` class, which is in the `javax.swing` package, to create a combo box. You pass an array of objects that are to be displayed as the items in the drop-down list to the constructor. Here is an example:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
```

When displayed, the combo box created by this code will initially appear as the button shown on the left in Figure 24-11. The button displays the item that is currently selected. Notice that the first item in the list is automatically selected when the combo box is first displayed. When the user clicks the button, the drop-down list appears and the user may select another item.

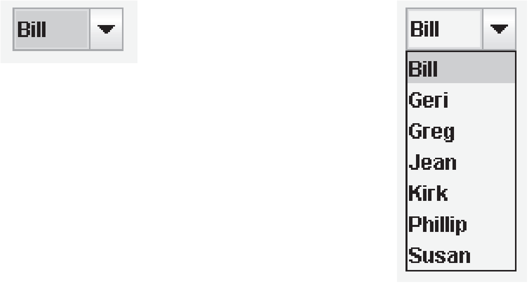


**Figure 24-11** A combo box (Oracle Corporate Counsel)

The combo box initially appears as a button that displays the selected item.

When the user clicks on the button, the list of items drops down. The user may select another item from the list.

As you can see, a combo box is a combination of two components. In the case of the combo box shown in Figure 24-11, it is the combination of a button and a list. This is where the name “combo box” comes from.

### Responding to Combo Box Events

When an item in a `JComboBox` object is selected, it generates an action event. As with `JButton` components, you handle action events with an action event listener class, which must have an `actionPerformed` method. When the user selects an item in a combo box, the combo box executes its action event listener’s `actionPerformed` method, passing an `ActionEvent` object as an argument.

## Retrieving the Selected Item

There are two methods in the `JComboBox` class that you can use to determine which item in a combo box is currently selected: `getSelectedItem` and `getSelectedIndex`. The `getSelectedItem` method returns a reference to the item that is currently selected. For example, assume that `nameBox` references the `JComboBox` component shown earlier in Figure 24-11. The following code retrieves a reference to the name that is currently selected and assigns it to the `selectedName` variable:

```
String selectedName;
selectedName = (String) nameBox.getSelectedItem();
```

Note that the return value of the `getSelectedItem` method is an `Object` reference. In this code we had to cast the return value to the `String` type to store it in the `selectedName` variable.

The `getSelectedIndex` method returns the index of the selected item. As with `JList` components, the items that are stored in a combo box are numbered with indices that start at 0. You can use the index of the selected item to retrieve the item from an array. For example, assume that the following code was used to build the `nameBox` component shown in Figure 24-11:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
```

Because the `names` array holds the values displayed in the `namesBox` component, the following code could be used to determine the selected item:

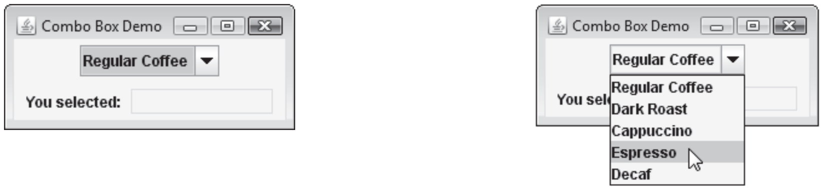
```
int index;
String selectedName;
index = nameList.getSelectedIndex();
selectedName = names[index];
```

The `ComboBoxWindow` class shown in Code Listing 24-4 demonstrates a combo box. It uses a `JComboBox` component with an action listener. When an item is selected from the combo box, it is displayed in a read-only text field. The `main` method creates an instance of the class, which initially displays the window shown at the top left of Figure 24-12. When the user clicks the combo box button, the drop-down list appears as shown in the top right of the figure. After the user selects Espresso from the list, the window appears as shown at the bottom of the figure.

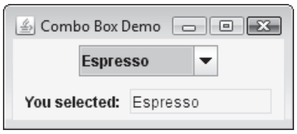
**Figure 24-12** The window displayed by the `ComboBoxWindow` class (Oracle Corporate Counsel)

This is the window that initially appears.

When the user clicks on the combo box button, the drop-down list appears.


Combo Box Demo
Regular Coffee
You selected:
Combo Box Demo
Regular Coffee
You sel
Regular Coffee
Dark Roast
Cappuccino
Espresso
Decaf

The item selected by the user appears in the read-only text field.


Combo Box Demo
Espresso
You selected:
Espresso

## Code Listing 24-4 (`ComboBoxWindow.java`)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   This class demonstrates a combo box.
*/

public class ComboBoxWindow extends JFrame
{
   private JPanel coffeePanel;           // To hold components
   private JPanel selectedCoffeePanel;   // To hold components
   private JComboBox coffeeBox;          // A list of coffees
   private JLabel label;                 // Displays a message
   private JTextField selectedCoffee;    // Selected coffee

   // The following array holds the values that will
   // be displayed in the coffeeBox combo box.
   private String[] coffee = { "Regular Coffee",
                               "Dark Roast", "Cappuccino",
                               "Espresso", "Decaf"};

   /**
      Constructor
   */

   public ComboBoxWindow()
   {
      // Set the title.
      setTitle("Combo Box Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildCoffeePanel();
      buildSelectedCoffeePanel();

      // Add the panels to the content pane.
      add(coffeePanel, BorderLayout.CENTER);
      add(selectedCoffeePanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildCoffeePanel method adds a combo box
      with the types of coffee to a panel.
   */

   private void buildCoffeePanel()
   {
      // Create a panel to hold the combo box.
      coffeePanel = new JPanel();

      // Create the combo box.
      coffeeBox = new JComboBox(coffee);

      // Register an action listener.
      coffeeBox.addActionListener(new ComboBoxListener());

      // Add the combo box to the panel.
      coffeePanel.add(coffeeBox);
   }

   /**
      The buildSelectedCoffeePanel method adds a
      read-only text field to a panel.
   */

   private void buildSelectedCoffeePanel()
   {
      // Create a panel to hold the components.
      selectedCoffeePanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the uneditable text field.
      selectedCoffee = new JTextField(10);
      selectedCoffee.setEditable(false);

      // Add the label and text field to the panel.
      selectedCoffeePanel.add(label);
      selectedCoffeePanel.add(selectedCoffee);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the combo box.
   */

   private class ComboBoxListener
                     implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Get the selected coffee.
         String selection =
                (String) coffeeBox.getSelectedItem();

         // Display the selected coffee in the text field.
         selectedCoffee.setText(selection);
      }
   }

   /**
      The main method creates an instance of the
      ComboBoxWindow class, which causes it to display
      its window.
   */

   public static void main(String[] args)
   {
      new ComboBoxWindow();
   }
}
```

### Editable Combo Boxes

There are two types of combo boxes: uneditable and editable. The default type of combo box is uneditable. An uneditable combo box combines a button with a list and allows the user to select items from its list only. This is the type of combo box used in the previous examples.

An editable combo box combines a text field and a list. In addition to selecting items from the list, the user may also type input into the text field. You make a combo box editable by calling the component's `setEditable` method, passing `true` as the argument. Here is an example:

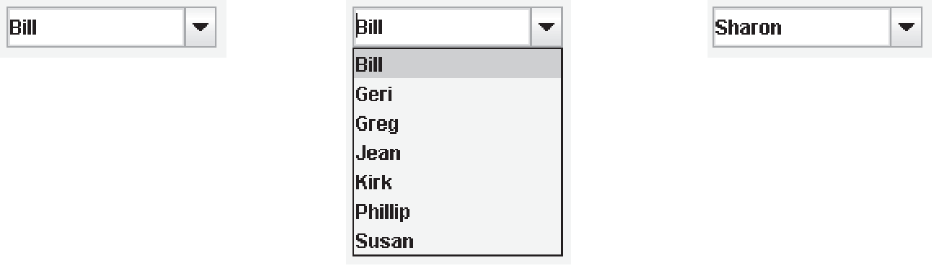
```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
nameBox.setEditable(true);
```

When displayed, the combo box created by this code initially appears as shown on the left of Figure 24-13. An editable combo box appears as a text field with a small button displaying an arrow joining it. The text field displays the item that is currently selected. When the user clicks the button, the drop-down list appears, as shown in the center of the figure. The user may select an item from the list. Alternatively, the user may type a value into the text field, as shown on the right of the figure. The user is not restricted to the values that appear in the list, and may type any input into the text field.

You can use the `getSelectedItem` method to retrieve a reference to the item that is currently selected. This method returns the item that appears in the combo box's text field, so it may or may not be an item that appears in the combo box's list.

The `getSelectedIndex` method returns the index of the selected item. However, if the user has entered a value in the text field that does not appear in the list, this method will return −1.

**Figure 24-13** An editable combo box (Oracle Corporate Counsel)



The editable combo box initially appears as a text field that displays the selected item. A small button with an arrow appears next to the text field.

When the user clicks on the button, the list of items drops down. The user may select another item from the list.

Alternatively, the user may type input into the text field. The user may type a value that does not appear in the list.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

24.1 How do you make a text field read-only? In code, how do you store text in a text field?

24.2 What is the index of the first item stored in a `JList` or a `JComboBox` component? If one of these components holds 12 items, what is the index of the 12th item?

24.3 How do you retrieve the selected item from a `JList` component? How do you get the index of the selected item?

24.4 How do you cause a scroll bar to be displayed with a `JList` component?

24.5 How do you retrieve the selected item from a `JComboBox` component? How do you get the index of the selected item?

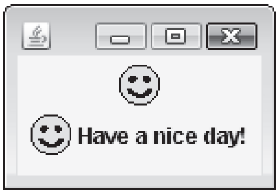
24.6 What is the difference between an uneditable and an editable combo box? Which of these is a combo box by default?

## 24.5 Displaying Images in Labels and Buttons

**CONCEPT: Images may be displayed in labels and buttons. You use the `ImageIcon` class to get an image from a file.**

In addition to displaying text in a label, you can also display an image. For example, Figure 24-14 shows a window with two labels. The top label displays a smiley face image and no text. The bottom label displays a smiley face image and text.

**Figure 24-14** Labels displaying an image icon (Oracle Corporate Counsel)



To display an image, first you create an instance of the `ImageIcon` class, which can read the contents of an image file. The `ImageIcon` class is part of the `javax.swing` package. The constructor accepts a `String` argument that is the name of an image file. The supported file types are JPEG, GIF, and PNG. The name can also contain path information. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
```

This statement creates an `ImageIcon` object that reads the contents of the file *Smiley.gif*. Because no path was given, it is assumed that the file is in the current directory or folder. Here is an example that uses a path:

```
ImageIcon image = new ImageIcon("C:\\Chapter 24\\Images\\Smiley.gif");
```

Next, you can display the image in a label by passing the `ImageIcon` object as an argument to the `JLabel` constructor. Here is the general format of the constructor:

```
JLabel(Icon image)
```

The argument passed to the image parameter can be an `ImageIcon` object or any object that implements the `Icon` interface. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
JLabel label = new JLabel(image);
```

This creates a label with an image, but no text. You can also create a label with both an image and text. An easy way to do this is to create the label with text, as usual, and then use the `JLabel` class's `setIcon` method to add an image to the label. The `setIcon` method accepts an `ImageIcon` object as its argument. Here is an example:

```
JLabel label = new JLabel("Have a nice day!");
label.setIcon(image);
```

The text will be displayed to the right of the image. The `JLabel` class also has the following constructor:

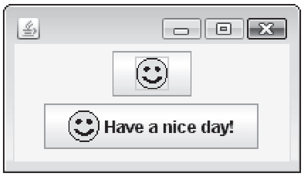
```
JLabel(String text, Icon image, int horizontalAlignment)
```

The first argument is the text to be displayed, the second argument is the image to be displayed, and the third argument is an `int` that specifies the horizontal alignment of the label contents. You should use the constants `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT` to specify the horizontal alignment. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
JLabel label = new JLabel("Have a nice day!",
                          image,
                          SwingConstants.RIGHT);
```

You can also display images in buttons, as shown in Figure 24-15.

**Figure 24-15** Buttons displaying an image icon (Oracle Corporate Counsel)



The process of creating a button with an image is similar to that of creating a label with an image. You use an `ImageIcon` object to read the image file, then pass the `ImageIcon` object as an argument to the `JButton` constructor. To create a button with an image and no text, pass only the `ImageIcon` object to the constructor. Here is an example:

```
// Create a button with an image, but no text.
ImageIcon image = new ImageIcon("Smiley.gif");
JButton button = new JButton(image);
```

To create a button with an image and text, pass a `String` and an `ImageIcon` object to the constructor. Here is an example:

```
// Create a button with an image and text.
ImageIcon image = new ImageIcon("Smiley.gif");
JButton button = new JButton("Have a nice day!", image);
```

To add an image to an existing button, pass an `ImageIcon` object to the button's `setIcon` method. Here is an example:

```
// Create a button with an image and text.
JButton button = new JButton("Have a nice day!");
ImageIcon image = new ImageIcon("Smiley.gif");
button.setIcon(image);
```

You are not limited to small graphical icons when placing images in labels or buttons. For example, the `MyCatImage` class in Code Listing 24-5 displays a digital photograph in a label when the user clicks a button. The `main` method creates an instance of the class, which displays the window shown at the left in Figure 24-16. When the user clicks the Get Image button, the window displays the image shown at the right in the figure.

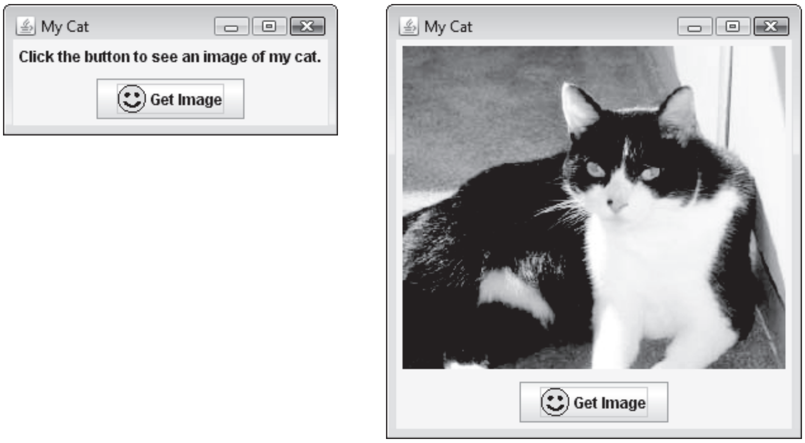
**Code Listing 24-5** (`MyCatImage.java`)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   This class demonstrates how to use an ImageIcon
   and a JLabel to display an image.
*/

```

```
public class MyCatImage extends JFrame
{
   private JPanel imagePanel;     // To hold the label
   private JPanel buttonPanel;    // To hold a button
   private JLabel imageLabel;     // To show an image
   private JButton button;        // To get an image


   /**
      Constructor
   */

   public MyCatImage()
   {
      // Set the title.
      setTitle("My Cat");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildImagePanel();
      buildButtonPanel();

      // Add the panels to the content pane.
      add(imagePanel, BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildImagePanel method adds a label to a panel.
   */

   private void buildImagePanel()
   {
      // Create a panel.
      imagePanel = new JPanel();

      // Create a label.
      imageLabel = new JLabel("Click the button to " +
                              "see an image of my cat.");

      // Add the label to the panel.
      imagePanel.add(imageLabel);
   }

   /**
      The buildButtonPanel method adds a button
      to a panel.
   */

   private void buildButtonPanel()
   {
      ImageIcon smileyImage;

      // Create a panel.
      buttonPanel = new JPanel();

      // Get the smiley face image.
      smileyImage = new ImageIcon("Smiley.gif");

      // Create a button.
      button = new JButton("Get Image");
      button.setIcon(smileyImage);

      // Register an action listener with the button.
      button.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(button);
   }

   /**
      Private inner class that handles the event when
      the user clicks the button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Read the image file into an ImageIcon object.
         ImageIcon catImage = new ImageIcon("Cat.jpg");

         // Display the image in the label.
         imageLabel.setIcon(catImage);

         // Remove the text from the label.
         imageLabel.setText(null);
```

```
106
107          // Pack the frame again to accommodate the
108          // new size of the label.
109          pack();
110       }
111    }
112
113    /**
114       The main method creates an instance of the
115       MyCatImage class, which causes it to display
116       its window.
117    */
118    public static void main(String[] args)
119    {
120       new MyCatImage();
121    }
122 }
```

**Figure 24-16** Window displayed by the `MyCatImage` class (Oracle Corporate Counsel)



Click the button to see an image of my cat.

This window initially appears.

When the user clicks the Get Image button, this image appears.

Let's take a closer look at the `MyCatImage` class. After some initial setup, the constructor calls the `buildImagePanel` method in line 34. Inside the `buildImagePanel` method, line 53 creates a `JPanel` component, referenced by the `imagePanel` variable, and then lines 56 and 57 create a `JLabel` component, referenced by the `imageLabel` variable. This is the label that will display the image when the user clicks the button. The last statement in the method, in line 60, adds the `imageLabel` component to the `imagePanel` panel.

Back in the constructor, line 35 calls the `buildButtonPanel` method, which creates the Get Image button and adds it to a panel. An instance of the `ButtonListener` inner class is also registered as the button's action listener. Let's look at the `ButtonListener` class's `actionPerformed` method. This method is executed when the user clicks the Get Image button. First, in line 99, an `ImageIcon` object is created from the file *Cat.jpg*. This file is in the same directory as the class. Next, in line 102, the image is stored in the `imageLabel` component. In line 105 the text that is currently displayed in the label is removed by passing `null` to the `imageLabel` component's `setText` method. The last statement, in line 109, calls the `JFrame` class's `pack` method. When the image was loaded into the `JLabel` component, the component resized itself to accommodate its new contents. The `JFrame` that encloses the window does not automatically resize itself, so we must call the `pack` method. This forces the `JFrame` to resize itself.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

24.7 How do you store an image in a `JLabel` component? How do you store both an image and text in a `JLabel` component?

24.8 How do you store an image in a `JButton` component? How do you store both an image and text in a `JButton` component?

24.9 What method do you use to store an image in an existing `JLabel` or `JButton` component?

## 24.6 Mnemonics and Tool Tips

**CONCEPT: A mnemonic is a key that you press while holding down the Alt key to interact with a component. A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component.**

### Mnemonics

A mnemonic is a key on the keyboard that you press in combination with the Alt key to access a component such as a button quickly. These are sometimes referred to as shortcut keys, or hot keys. When you assign a mnemonic to a button, the user can click the button by holding down the Alt key and pressing the mnemonic key. Although users can interact with components with either the mouse or their mnemonic keys, those who are quick with the keyboard usually prefer to use mnemonic keys instead of the mouse.

You assign a mnemonic to a component through the component's `setMnemonic` method, which is inherited from the `AbstractButton` class. The method's general format is as follows:

```
void setMnemonic(int key)
```

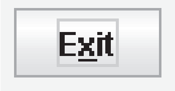
The argument that you pass to the method is an integer code that represents the key you wish to assign as a mnemonic. The `KeyEvent` class, which is in the `java.awt.event` package, has predefined constants that you can use. These constants take the form `KeyEvent.VK_x`, where `x` is a key on the keyboard. For example, to assign the A key as a mnemonic, you would use `KeyEvent.VK_A`. (The letters VK in the constants stand for "virtual key".) Here is an example of code that creates a button with the text "Exit" and assigns the X key as the mnemonic:

```
JButton exitButton = new JButton("Exit");
exitButton.setMnemonic(KeyEvent.VK_X);
```

The user may click this button by pressing a +X on the keyboard. (This means holding down the Alt key and pressing X.)

If the letter chosen as the mnemonic is in the component's text, the first occurrence of that letter will appear underlined when the component is displayed. For example, the button created with the previous code has the text "Exit". Because X was chosen as the mnemonic, the letter x will appear underlined, as shown in Figure 24-17.

**Figure 24-17** Button with mnemonic X (Oracle Corporate Counsel)



If the mnemonic is a letter that does not appear in the component's text, then no letter will appear underlined.

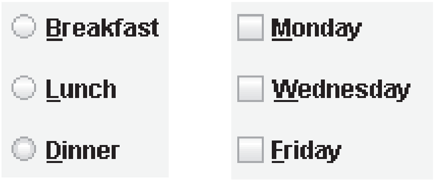
**NOTE:** The `KeyEvent` class also has constants for symbols. For example, the constant for the `!` symbol is `VK_EXCLAMATION_MARK`, and the constant for the `&` symbol is `VK_AMPERSAND`. See the Java API documentation for the `KeyEvent` class for a list of all the constants.

You can also assign mnemonics to radio buttons and check boxes, as shown in the following code:

```
//Create three radio buttons and assign mnemonics.
JRadioButton rb1 = new JRadioButton("Breakfast");
rb1.setMnemonic(KeyEvent.VK_B);
JRadioButton rb2 = new JRadioButton("Lunch");
rb2.setMnemonic(KeyEvent.VK_L);
JRadioButton rb3 = new JRadioButton("Dinner");
rb3.setMnemonic(KeyEvent.VK_D);

// Create three check boxes and assign mnemonics.
JCheckBox cb1 = new JCheckBox("Monday");
cb1.setMnemonic(KeyEvent.VK_M);
JCheckBox cb2 = new JCheckBox("Wednesday");
cb2.setMnemonic(KeyEvent.VK_W);
JCheckBox cb3 = new JCheckBox("Friday");
cb3.setMnemonic(KeyEvent.VK_F);
```

This code will create the components shown in Figure 24-18.

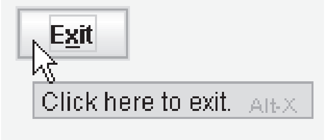
**Figure 24-18** Radio buttons and check boxes with mnemonics assigned (Oracle Corporate Counsel)



## Tool Tips

A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component. The box usually gives a short description of what the component does. Most GUI applications use tool tips as a way of providing immediate and concise help to the user. For example, Figure 24-19 shows a button with its tool tip displayed.

**Figure 24-19** Button with tool tip displayed (Oracle Corporate Counsel)



You assign a tool tip to a component with the `setToolTipText` method, which is inherited from the `JComponent` class. Here is the method's general format:

```
void setToolTipText(String text)
```

The `String` that is passed as an argument is the text that will be displayed in the component's tool tip. For example, the following code creates the Exit button shown in Figure 24-19 and its associated tool tip:

```
JButton exitButton = new JButton("Exit");
exitButton.setToolTipText("Click here to exit.");
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

24.10 What is a mnemonic? How do you assign a mnemonic to a component?

24.11 What is a tool tip? How do you assign a tool tip to a component?

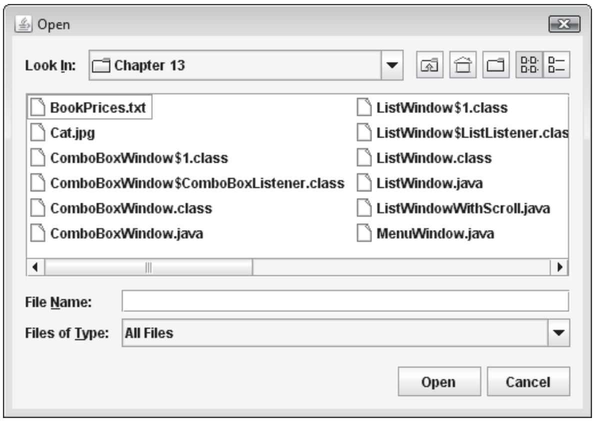
## 24.7 File Choosers and Color Choosers

**CONCEPT: Java provides components that equip your applications with standard dialog boxes for opening files, saving files, and selecting colors.**

## File Choosers

A file chooser is a specialized dialog box that allows the user to browse for a file and select it. Figure 24-20 shows an example of a file chooser dialog box.

**Figure 24-20** A file chooser dialog box for opening a file (Oracle Corporate Counsel)



Look In: Chapter 13

File Name:

Files of Type: All Files

Open
BookPrices.txt
Cat.jpg
ComboBoxWindow$1.class
ComboBoxWindow$ComboBoxListener.class
ComboBoxWindow.class
ComboBoxWindow.java
ListWindow$1.class
ListWindow$ListListener.clas
ListWindow.class
ListWindow.java
ListWindowWithScroll.java
MenuWindow.java
Open
Cancel

You create an instance of the `JFileChooser` class, which is part of the `javax.swing` package, to display a file chooser dialog box. The class has several constructors. We will focus on two of them, which have the following general formats:

```
JFileChooser()
JFileChooser(String path)
```

The first constructor shown takes no arguments. This constructor uses the default directory as the starting point for all of its dialog boxes. If you are using Windows, this will probably be the "My Documents" folder under your account. If you are using UNIX, this will be your login directory. The second constructor takes a `String` argument containing a valid path. This path will be the starting point for the object's dialog boxes.

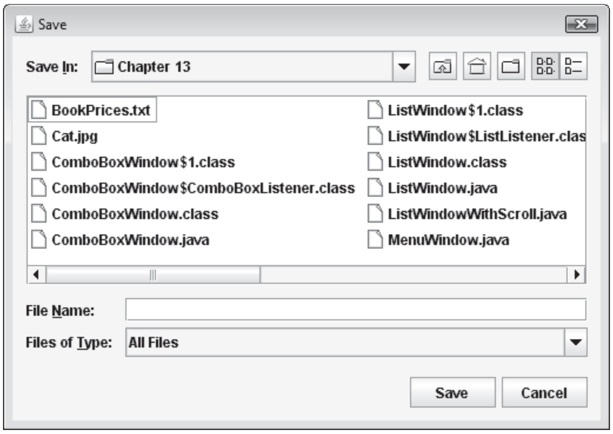
A `JFileChooser` object can display two types of predefined dialog boxes: an open file dialog box and a save file dialog box. Figure 24-20 shows an example of an open file dialog box. It lets the user browse for an existing file to open. A save file dialog box, as shown in Figure 24-21, is employed when the user needs to browse to a location to save a file. Both of these dialog boxes appear the same, except the open file dialog box displays "Open" in its title bar, and the save file dialog box displays "Save." Also, the open file dialog box has an Open button, and the save file dialog box has a Save button. There is no difference in the way they operate.

### Displaying a File Chooser Dialog Box

To display an open file dialog box, use the `showOpenDialog` method. The method's general format is as follows:

```
int showOpenDialog(Component parent)
```

**Figure 24-21** A save file dialog box (Oracle Corporate Counsel)


Save
BookPrices.txt
Cat.jpg
ComboBoxWindow$1.class
ComboBoxWindow$ComboBoxListener.class
ComboBoxWindow.class
ComboBoxWindow.java
ListWindow$1.class
ListWindow$ListListener.clas
ListWindow.class
ListWindow.java
ListWindowWithScroll.java
MenuWindow.java
File Name:
Save
Cancel

Save In: Chapter 13

Files of Type: All Files

The argument can be either `null` or a reference to a component. If you pass `null`, the dialog box is normally centered in the screen. If you pass a reference to a component, such as `JFrame`, the dialog box is displayed over the component.

To display a save file dialog box, use the `showSaveDialog` method. The method's general format is as follows:

```
int showSaveDialog(Component parent)
```

Once again, the argument can be either `null` or a reference to a component. Both the `showOpenDialog` and `showSaveDialog` methods return an integer that indicates the action taken by the user to close the dialog box. You can compare the return value to one of the following constants:

- **`JFileChooser.CANCEL_OPTION`.** This return value indicates that the user clicked the Cancel button.
- **`JFileChooser.APPROVE_OPTION`.** This return value indicates that the user clicked the Open or Save button.
- **`JFileChooser.ERROR_OPTION`.** This return value indicates that an error occurred, or the user clicked the standard close button on the window to dismiss it.

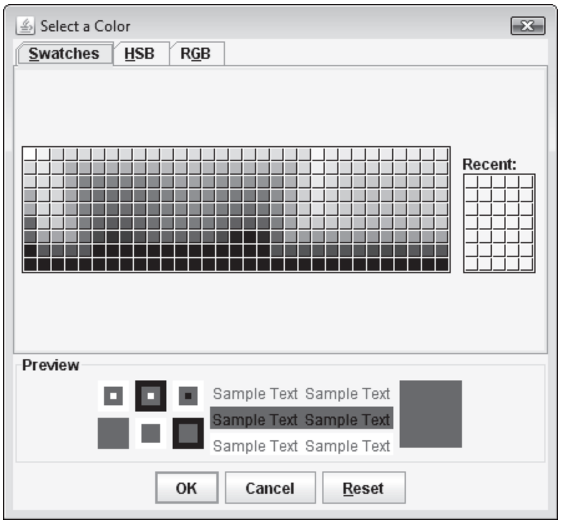
If the user selected a file, you can use the `getSelectedFile` method to determine the file that was selected. The `getSelectedFile` method returns a `File` object, which contains data about the selected file. The `File` class is part of the `java.io` package. You can use the `File` object's `getPath` method to get the path and file name as a `String`. Here is an example:

```
JFileChooser fileChooser = new JFileChooser();
int status = fileChooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION)
{
   File selectedFile = fileChooser.getSelectedFile();
   String filename = selectedFile.getPath();
   JOptionPane.showMessageDialog(null, "You selected " + filename);
}
```

## Color Choosers

A color chooser is a specialized dialog box that allows the user to select a color from a predefined palette of colors. Figure 24-22 shows an example of a color chooser. By clicking the HSB tab you can select a color by specifying its hue, saturation, and brightness. By clicking the RGB tab you can select a color by specifying its red, green, and blue components.

**Figure 24-22** A color chooser dialog box (Oracle Corporate Counsel)



You use the `JColorChooser` class, which is part of the `javax.swing` package, to display a color chooser dialog box. You do not create an instance of the class, however. It has a static method named `showDialog`, with the following general format:

```
Color showDialog(Component parent, String title, Color initial)
```

The first argument can be either `null` or a reference to a component. If you pass `null`, the dialog box is normally centered in the screen. If you pass a reference to a component, such as `JFrame`, the dialog box is displayed over the component. The second argument is text that is displayed in the dialog box's title bar. The third argument indicates the color that appears initially selected in the dialog box. This method returns the color selected by the user. The following code is an example. This code allows the user to select a color, and then that color is assigned as a panel's background color.

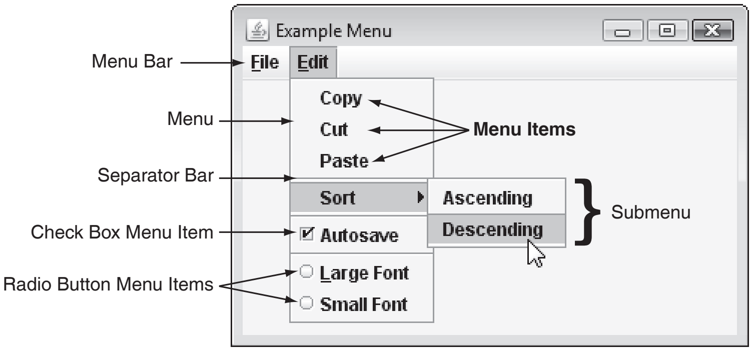
```
JPanel panel = new JPanel();
Color selectedColor;
selectedColor = JColorChooser.showDialog(null,
                "Select a Background Color", Color.BLUE);
panel.setBackground(selectedColor);
```

## 24.8 Menus

**CONCEPT: Java provides classes for creating systems of drop-down menus. Menus can contain menu items, checked menu items, radio button menu items, and other menus.**

In the GUI applications you have studied so far, the user initiates actions by clicking components such as buttons. When an application has several operations for the user to choose from, a menu system is more commonly used than buttons. A menu system is a collection of commands organized in one or more drop-down menus. Before learning how to construct a menu system, you must learn about the basic items that are found in a typical menu system. Look at the example menu system in Figure 24-23.

**Figure 24-23** Example menu system (Oracle Corporate Counsel)



The menu system in the figure consists of the following items:

- **Menu Bar.** At the top of the window, just below the title bar, is a menu bar. The menu bar lists the names of one or more menus. The menu bar in Figure 24-23 shows the names of two menus: File and Edit.
- **Menu.** A menu is a drop-down list of menu items. The user may activate a menu by clicking on its name on the menu bar. In the figure, the Edit menu has been activated.
- **Menu Item.** A menu item can be selected by the user. When a menu item is selected, some type of action is usually performed.
- **Check box menu item.** A check box menu item appears with a small box beside it. The item may be selected or deselected. When it is selected, a check mark appears in the box. When it is deselected, the box appears empty. Check box menu items are normally used to turn an option on or off. The user toggles the state of a check box menu item each time he or she selects it.
- **Radio button menu item.** A radio button menu item may be selected or deselected. A small circle appears beside it that is filled in when the item is selected and empty when the item is deselected. Like a check box menu item, a radio button menu item can be used to turn an option on or off. When a set of radio button menu items are grouped

with a `ButtonGroup` object, only one of them can be selected at a time. When the user selects a radio button menu item, the one that was previously selected is deselected.

- **Submenu.** A menu within a menu is called a submenu. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear. For example, in Figure 24-23, clicking on the Sort command causes a submenu to appear.
- **Separator bar.** A separator bar is a horizontal bar that is used to separate groups of items on a menu. Separator bars are only used as a visual aid and cannot be selected by the user.

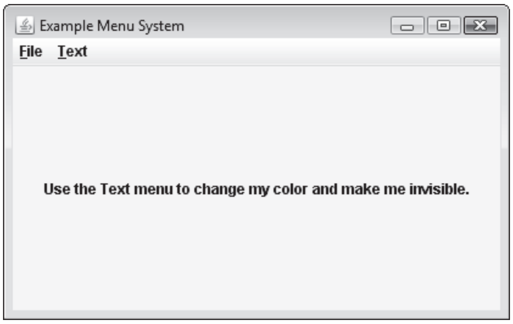
A menu system is constructed with the following classes:

- **`JMenuItem`.** Use this class to create a regular menu item. A `JMenuItem` component generates an action event when the user selects it.
- **`JCheckBoxMenuItem`.** Use this class to create a check box menu item. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `JCheckBoxMenuItem` component generates an action event when the user selects it.
- **`JRadioButtonMenuItem`.** Use this class to create a radio button menu item. `JRadioButtonMenuItem` components can be grouped in a `ButtonGroup` object so that only one of them can be selected at a time. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `JRadioButtonMenuItem` component generates an action event when the user selects it.
- **`JMenu`.** Use this class to create a menu. A `JMenu` component can contain `JMenuItem`, `JCheckBoxMenuItem`, and `JRadioButton` components, as well as other `JMenu` components. A submenu is a `JMenu` component that is inside another `JMenu` component.
- **`JMenuBar`.** Use this class to create a menu bar. A `JMenuBar` object can contain `JMenu` components.

All of these classes are in the `javax.swing` package. A menu system is a `JMenuBar` component that contains one or more `JMenu` components. Each `JMenu` component can contain `JMenuItem`, `JRadioButtonMenuItem`, and `JCheckBoxMenuItem` components, as well as other `JMenu` components. The classes contain all of the code necessary to operate the menu system.

To see an example of an application that uses a menu system, we look at the `MenuWindow` class shown in Code Listing 24-6. The class displays the window shown in Figure 24-24.

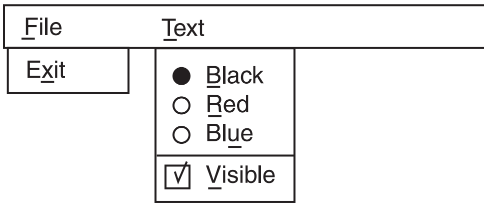
**Figure 24-24** Window displayed by the `MenuWindow` class (Oracle Corporate Counsel)



Use the Text menu to change my color and make me invisible.

The class demonstrates how a label appears in different colors. Notice that the window has a menu bar with two menus: File and Text. Figure 24-25 shows a sketch of the menu system. When the user opens the Text menu, he or she can select a color using the radio button menu items and the label will change to the selected color. The Text menu also contains a Visible item, which is a check box menu item. When this item is selected (checked), the label is visible. When this item is deselected (unchecked), the label is invisible.

**Figure 24-25** Sketch of the `MenuWindow` class's menu system (Oracle Corporate Counsel)



**Code Listing 24-6** (`MenuWindow.java`)

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   The MenuWindow class demonstrates a menu system.
*/

public class MenuWindow extends JFrame
{
   private JLabel messageLabel;             // Displays a message
   private final int LABEL_WIDTH = 400;     // Label's width
   private final int LABEL_HEIGHT = 200;    // Label's height

   // The following will reference menu components.
   private JMenuBar menuBar;                // The menu bar
   private JMenu fileMenu;                  // The File menu
   private JMenu textMenu;                  // The Text menu
   private JMenuItem exitItem;              // To exit
   private JRadioButtonMenuItem blackItem;  // Makes text black
   private JRadioButtonMenuItem redItem;    // Makes text red
   private JRadioButtonMenuItem blueItem;   // Makes text blue
   private JCheckBoxMenuItem visibleItem;   // Toggle visibility

   /**
      Constructor
   */

   public MenuWindow()
   {
      // Set the title.
      setTitle("Example Menu System");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the messageLabel label.
      messageLabel = new JLabel("Use the Text menu to " +
                "change my color and make me invisible.",
                SwingConstants.CENTER);

      // Set the label's preferred size.
      messageLabel.setPreferredSize(
             new Dimension(LABEL_WIDTH, LABEL_HEIGHT));

      // Set the label's foreground color.
      messageLabel.setForeground(Color.BLACK);

      // Add the label to the content pane.
      add(messageLabel);

      // Build the menu bar.
      buildMenuBar();

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMenuBar method builds the menu bar.
   */

   private void buildMenuBar()
   {
      // Create the menu bar.
      menuBar = new JMenuBar();

      // Create the file and text menus.
      buildFileMenu();
      buildTextMenu();

      // Add the file and text menus to the menu bar.
      menuBar.add(fileMenu);
      menuBar.add(textMenu);

      // Set the window's menu bar.
      setJMenuBar(menuBar);
   }

   /**
      The buildFileMenu method builds the File menu
      and returns a reference to its JMenu object.
   */

   private void buildFileMenu()
   {
      // Create an Exit menu item.
      exitItem = new JMenuItem("Exit");
      exitItem.setMnemonic(KeyEvent.VK_X);
      exitItem.addActionListener(new ExitListener());

      // Create a JMenu object for the File menu.
      fileMenu = new JMenu("File");
      fileMenu.setMnemonic(KeyEvent.VK_F);

      // Add the Exit menu item to the File menu.
      fileMenu.add(exitItem);
   }

   /**
      The buildTextMenu method builds the Text menu
      and returns a reference to its JMenu object.
   */

   private void buildTextMenu()
   {
      // Create the radio button menu items to change
      // the color of the text. Add an action listener
      // to each one.
      blackItem = new JRadioButtonMenuItem("Black", true);
      blackItem.setMnemonic(KeyEvent.VK_B);
      blackItem.addActionListener(new ColorListener());

      redItem = new JRadioButtonMenuItem("Red");
      redItem.setMnemonic(KeyEvent.VK_R);
      redItem.addActionListener(new ColorListener());

      blueItem = new JRadioButtonMenuItem("Blue");
      blueItem.setMnemonic(KeyEvent.VK_U);
      blueItem.addActionListener(new ColorListener());

      // Create a button group for the radio button items.
      ButtonGroup group = new ButtonGroup();
      group.add(blackItem);
      group.add(redItem);
      group.add(blueItem);

      // Create a check box menu item to make the text
      // visible or invisible.
      visibleItem = new JCheckBoxMenuItem("Visible", true);
      visibleItem.setMnemonic(KeyEvent.VK_V);
      visibleItem.addActionListener(new VisibleListener());

      // Create a JMenu object for the Text menu.
      textMenu = new JMenu("Text");
      textMenu.setMnemonic(KeyEvent.VK_T);

      // Add the menu items to the Text menu.
      textMenu.add(blackItem);
      textMenu.add(redItem);
      textMenu.add(blueItem);
      textMenu.addSeparator();    // Add a separator bar.
      textMenu.add(visibleItem);
   }

   /**
      Private inner class that handles the event that
      is generated when the user selects Exit from
      the File menu.
   */

   private class ExitListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0);
      }
   }

   /**
      Private inner class that handles the event that
      is generated when the user selects a color from
      the Text menu.
   */

   private class ColorListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         if (blackItem.isSelected())
            messageLabel.setForeground(Color.BLACK);
```

```
173          else if (redItem.isSelected())
174             messageLabel.setForeground(Color.RED);
175          else if (blueItem.isSelected())
176             messageLabel.setForeground(Color.BLUE);
177       }
178    }
179
180    /**
181       Private inner class that handles the event that
182       is generated when the user selects Visible from
183       the Text menu.
184    */
185
186    private class VisibleListener implements ActionListener
187    {
188       public void actionPerformed(ActionEvent e)
189       {
190          if (visibleItem.isSelected())
191             messageLabel.setVisible(true);
192          else
193             messageLabel.setVisible(false);
194       }
195    }
196
197    /**
198       The main method creates an instance of the
199       MenuWindow class, which causes it to display
200       its window.
201    */
202
203    public static void main(String[] args)
204    {
205       MenuWindow mw = new MenuWindow();
206    }
207 }
```

Let's take a closer look at the `MenuWindow` class. Before we examine how the menu system is constructed, we should explain the code in lines 38 through 44. Lines 38 through 40 create the `messageLabel` component and align its text in the label's center. Then, in lines 43 and 44, the `setPreferredSize` method is called. The `setPreferredSize` method is inherited from the `JComponent` class, and it establishes a component's preferred size. It is called the *preferred size* because the layout manager adjusts the component's size when necessary. Normally, a label's preferred size is determined automatically, depending on its contents. We want to make this label larger, however, so the window will be larger when it is packed around the label.

The `setPreferredSize` method accepts a `Dimension` object as its argument. A `Dimension` object specifies a component's width and height. The first argument to the `Dimension` class constructor is the component's width, and the second argument is the component's height. In this class, the `LABEL_WIDTH` and `LABEL_HEIGHT` constants are defined with the values 400 and 200 respectively. So, this statement sets the label's preferred size to 400 pixels wide by 200 pixels high. (The `Dimension` class is part of the `java.awt` package.) Notice from Figure 24-24 that this code does not affect the size of the text that is displayed in the label, only the size of the label component.

To create the menu system, the constructor calls the `buildMenuBar` method in line 53. Inside this method, the statement in line 67 creates a `JMenuBar` component and assigns its address to the `menuBar` variable. The `JMenuBar` component acts as a container for `JMenu` components. The menu bar in this application has two menus: File and Text.

Next, the statement in line 70 calls the `buildFileMenu` method. The `buildFileMenu` method creates the File menu, which has only one item: Exit. The statement in line 89 creates a `JMenuItem` component for the Exit item, which is referenced by the `exitItem` variable. The `String` that is passed to the `JMenuItem` constructor is the text that will appear on a menu for this menu item. The statement in line 90 assigns the x key as a mnemonic to the `exitItem` component. Then, line 91 creates an action listener for the component (an instance of `ExitListener`, a private inner class), which causes the application to end.

Next, line 94 creates a `JMenu` object for the File menu. Notice that the name of the menu is passed as an argument to the `JMenu` constructor. Line 95 assigns the F key to the File menu as a mnemonic. The last statement in the `buildFileMenu` method, in line 98, adds `exitItem` to the `fileMenu` component.

Back in the `buildMenuBar` method, the statement in line 71 calls the `buildTextMenu` method. The `buildTextMenu` method builds the Text menu, which has three radio button menu items (Black, Red, and Blue), a separator bar, and a check box menu item (Visible). The code in lines 111 through 121 creates the radio button menu items, assigns mnemonic keys to them, and adds an action listener to each.

The `JRadioButtonItem` constructor accepts a `String` argument, which is the menu item's text. By default, a radio button menu item is not initially selected. The constructor can also accept an optional second argument, which is a `boolean` value indicating whether the item should be initially selected. Notice that in line 111, `true` is passed as the second argument to the `JRadioButtonItem` constructor. This causes the Black menu item to be initially selected.

Next, in lines 124 through 127, a button group is created and the radio button menu items are added to it. As with `JRadioButton` components, `JRadioButtonMenuItem` components may be grouped in a `ButtonGroup` object. As a result, only one of the grouped menu items may be selected at a time. When one is selected, any other menu item in the group is deselected.

Next, the Visible item, a check box menu item, is created in line 131. Notice that `true` is passed as the second argument to the constructor. This causes the item to be initially selected. A mnemonic key is assigned in line 132, and an action listener is added to the component in line 133.

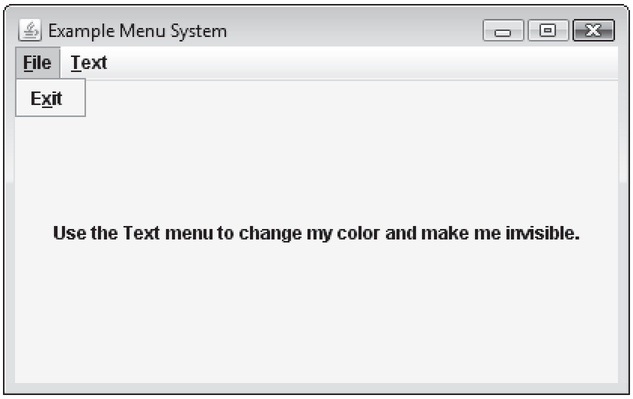
Line 136 creates a `JMenu` component for the Text menu, and line 137 assigns a mnemonic key to it. Lines 140 through 142 add the `blackItem`, `redItem`, and `blueItem` radio button menu items to the Text menu. In line 143, the `addSeparator` method is called to add a separator bar to the menu. Because the `addSeparator` method is called just after the `blueItem` component is added and just before the `visibleItem` component is added, it will appear between the Blue and Visible items on the menu. Line 144 adds the Visible item to the Text menu.

Back in the `buildMenuBar` method, in lines 74 and 75, the File menu and Text menu are added to the menu bar. In line 78, the `setJMenuBar` method is called, passing `menuBar` as an argument. The `setJMenuBar` method is a `JFrame` method that places a menu bar in a frame. You pass a `JMenuBar` component as the argument. When the `JFrame` is displayed, the menu bar will appear at its top.

Figure 24-26 shows how the class's window appears with the File menu and the Text menu opened. Selecting a color from the Text menu causes an instance of the `ColorListener` class to execute its `actionPerformed` method, which changes the color of the text. Selecting the Visible item causes an instance of the `VisibleListener` class to execute its `actionPerformed` method, which toggles the label's visibility.

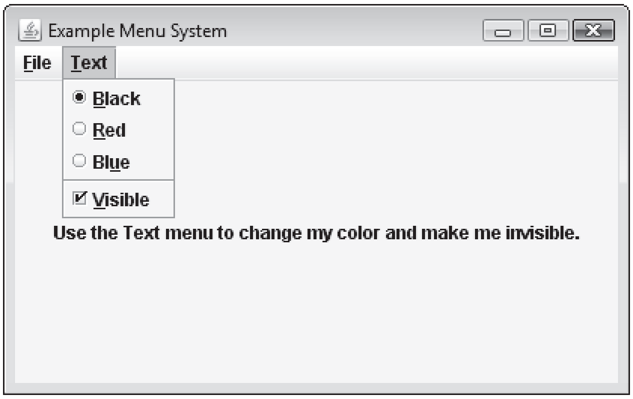
**Figure 24-26** The window with the File menu and Text menu opened (Oracle Corporate Counsel)

The window with the File menu opened.



Use the Text menu to change my color and make me invisible.

The window with the Text menu opened.



Use the Text menu to change my color and make me invisible.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

24.12 Briefly describe each of the following menu system items:
  a) Menu bar
  b) Menu item
  c) Check box menu item
  d) Radio button menu item
  e) Submenu
  f) Separator bar

24.13 What class do you use to create a regular menu item? What do you pass to the class constructor?

24.14 What class do you use to create a radio button menu item? What do you pass to the class constructor? How do you cause it to be initially selected?

24.15 How do you create a relationship between radio button menu items so that only one may be selected at a time?

24.16 What class do you use to create a check box menu item? What do you pass to the class constructor? How do you cause it to be initially selected?

24.17 What class do you use to create a menu? What do you pass to the class constructor?

24.18 What class do you use to create a menu bar?

24.19 How do you place a menu bar in a `JFrame`?

24.20 What type of event do menu items generate when selected by the user?

24.21 How do you change the size of a component such as a `JLabel` after it has been created?

24.22 What arguments do you pass to the `Dimension` class constructor?

## 24.9 More about Text Components: Text Areas and Fonts

**CONCEPT: A text area is a multi-line text field that can accept several lines of text input. Components that inherit from the `JComponent` class have a `setFont` method that allows you to change the font and style of the component's text.**

### Text Areas

In Chapter 23, you were introduced to the `JTextField` class, which is used to create text fields. A text field is a component that allows the user to enter a single line of text. A text area is like a text field that can accept multiple lines of input. You use the `JTextArea` class to create a text area. Here is the general format of two of the class's constructors:

```
JTextArea(int rows, int columns)
JTextArea(String text, int rows, int columns)
```

In both constructors, `rows` is the number of rows or lines of text that the text area is to display, and `columns` is the number of columns or characters that are to be displayed per line. In the second constructor, `text` is a string that the text area will initially display. For example, the following statement creates a text area with 20 rows and 40 columns:

```
JTextArea textInput = new JTextArea(20, 40);
```

The following statement creates a text area with 20 rows and 40 columns that will initially display the text stored in the `String` object `info`:

```
JTextArea textInput = new JTextArea(info, 20, 40);
```

As with the `JTextField` class, the `JTextArea` class provides the `getText` and `setText` methods for getting and setting the text contained in the component. For example, the following statement gets the text stored in the `textInput` text area and stores it in the `String` object `userText`:

```
String userText = textInput.getText();
```

The following statement stores the text that is in the `String` object `info` in the `textInput` text area:

```
textInput.setText(info);
```

`JTextArea` components do not automatically display scroll bars. To display scroll bars on a `JTextArea` component, you must add it to the scroll pane. As you already know, you create a scroll pane with the `JScrollPane` class. Here is an example of code that creates a text area and adds it to a scroll pane:

```
JTextArea textInput = new JTextArea(20, 40);
JScrollPane scrollPane = new JScrollPane(textInput);
```

The `JScrollPane` object displays both vertical and horizontal scroll bars on a text area. By default, the scroll bars are not displayed until they are needed; however, you can alter this behavior with two of the `JScrollPane` class's methods. The `setHorizontalScrollBarPolicy` method takes an `int` argument that specifies when a horizontal scroll bar should appear in the scroll pane. You can pass one of the following constants as an argument:

- **`JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED.`** This is the default setting. A horizontal scroll bar is displayed only when there is not enough horizontal space to display the text contained in the text area.
- **`JScrollPane.HORIZONTAL_SCROLLBAR_NEVER.`** This setting prevents a horizontal scroll bar from being displayed in the text area.
- **`JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS.`** With this setting, a horizontal scroll bar is always displayed, even when it is not needed.

The `setVerticalScrollBarPolicy` method also takes an `int` argument, which specifies when a vertical scroll bar should appear in the scroll pane. You can pass one of the following constants as an argument:

- **`JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED.`** This is the default setting. A vertical scroll bar is displayed only when there is not enough vertical space to display the text contained in the text area.

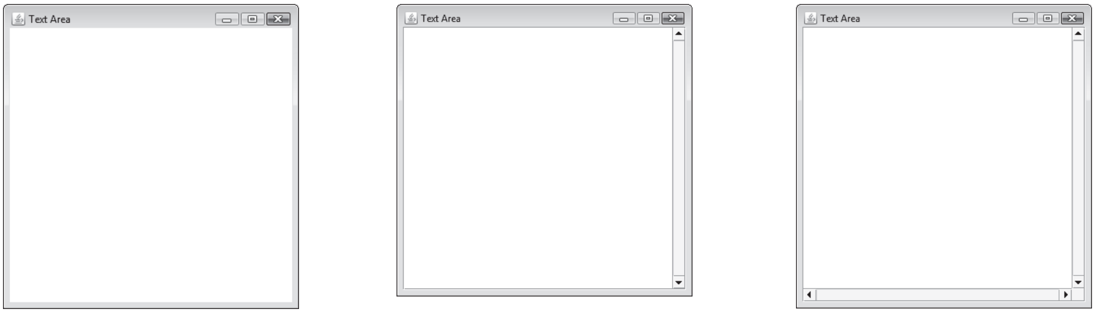
- **`JScrollPane.VERTICAL_SCROLLBAR_NEVER.`** This setting prevents a vertical scroll bar from being displayed in the text area.
- **`JScrollPane.VERTICAL_SCROLLBAR_ALWAYS.`** With this setting, a vertical scroll bar is always displayed, even when it is not needed.

For example, the following code specifies that a vertical scroll bar should always appear on a scroll pane's component, but a horizontal scroll bar should not appear:

```
scrollPane.setHorizontalScrollBarPolicy(
               JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
scrollPane.setVerticalScrollBarPolicy(
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

Figure 24-27 shows a text area without scroll bars, a text area with a vertical scroll bar, and a text area with both a horizontal and a vertical scroll bar.

**Figure 24-27** Text areas with and without scroll bars (Oracle Corporate Counsel)


A text area with no scroll bars.
A text area with a vertical scroll bar.
A text area with a vertical and a horizontal scroll bar.

By default, `JTextArea` components do not perform line wrapping. This means that when text is entered into the component and the end of a line is reached, the text does not wrap around to the next line. If you want line wrapping, you use the `JTextArea` class's `setLineWrap` method to turn it on. The method accepts a `boolean` argument. If you pass `true`, line wrapping is turned on. If you pass `false`, line wrapping is turned off. Here is an example of a statement that turns a text area's line wrapping on:

```
textInput.setLineWrap(true);
```

There are two different styles of line wrapping: word wrapping and character wrapping. When word wrapping is performed, the line breaks always occur between words, never in the middle of a word. When character wrapping is performed, lines are broken between characters. This means that lines can be broken in the middle of a word. You specify the style of line wrapping that you prefer with the `JTextArea` class's `setWrapStyleWord` method. This method accepts a `boolean` argument. If you pass `true`, the text area will perform word wrapping. If you pass `false`, the text area will perform character wrapping. The default style is character wrapping.

## Fonts

The appearance of a component's text is determined by the text's font, style, and size. The font is the name of the typeface—the style can be plain, bold, and/or italic—and the size is the size of the text in points. To change the appearance of a component's text you use the component's `setFont` method, which is inherited from the `JComponent` class. The general format of the method is as follows:

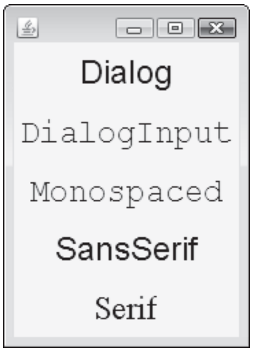
```
void setFont(Font appearance)
```

You pass a `Font` object as an argument to this method. The `Font` class constructor has the following general format:

```
Font(String fontName, int style, int size);
```

The first argument is the name of a font. Although the fonts that are available vary from system to system, Java guarantees that you will have Dialog, DialogInput, Monospaced, SansSerif, and Serif. Figure 24-28 shows an example of each of these.

**Figure 24-28** Examples of fonts (Oracle Corporate Counsel)



Dialog

DialogInput

Monospaced

SansSerif

Serif

The second argument to the `Font` constructor is an `int` that represents the style of the text. The `Font` class provides the following constants that you can use: `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. The third argument is the size of the text in points. (There are 72 points per inch, so a 72-point font has a height of one inch. Ten- and twelve-point fonts are normally used for most applications.) Here is an example of a statement that changes the text of a label to a 24-point bold serif font:

```
label.setFont(new Font("Serif", Font.BOLD, 24));
```

You can combine styles by mathematically adding them. For example, the following statement changes a label's text to a 24-point bold and italic serif font:

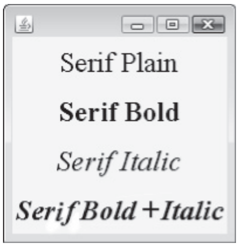
```
label.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

Figure 24-29 shows an example of the serif font in plain, bold, italic, and bold plus italic styles. The following code was used to create the labels:

```
JLabel label1 = new JLabel("Serif Plain", SwingConstants.CENTER);
label1.setFont(new Font("Serif", Font.PLAIN, 24));

JLabel label2 = new JLabel("Serif Bold", SwingConstants.CENTER);
label2.setFont(new Font("Serif", Font.BOLD, 24));
```

```
JLabel label3 = new JLabel("Serif Italic", SwingConstants.CENTER);
label3.setFont(new Font("Serif", Font.ITALIC, 24));

JLabel label4 = new JLabel("Serif Bold + Italic",
                           SwingConstants.CENTER);
label4.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

**Figure 24-29** Examples of serif plain, bold, italic, and bold plus italic (Oracle Corporate Counsel)



Serif Plain

**Serif Bold**

*Serif Italic*

***Serif Bold +Italic***

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

24.23 What arguments do you pass to the `JTextArea` constructor?

24.24 How do you retrieve the text that is stored in a `JTextArea` component?

24.25 Does the `JTextArea` component automatically display scroll bars? If not, how do you accomplish this?

24.26 What is line wrapping? What are the two styles of line wrapping? How do you turn a `JTextArea` component's line wrapping on? How do you select a line wrapping style?

24.27 What type of argument does a component's `setFont` method accept?

24.28 What are the arguments that you pass to the `Font` class constructor?

See the Simple Text Editor Application case study on this book's companion Web site (www.pearson.com/gaddis) for an in-depth example that uses menus and other topics from this chapter.

## 24.10 Sliders

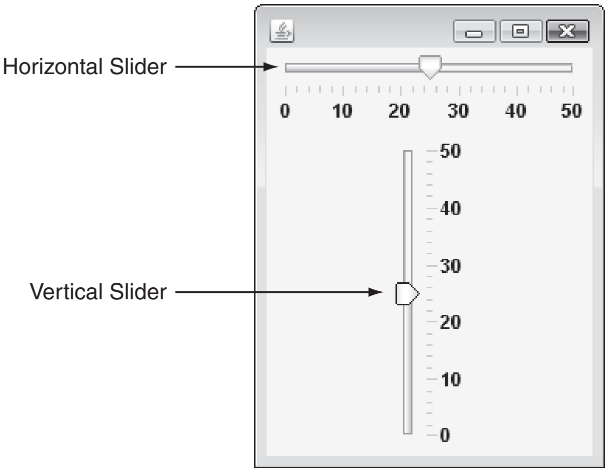
**CONCEPT: A slider is a component that allows the user to adjust a number graphically within a range of values.**

Sliders, which are created from the `JSlider` class, display an image of a "slider knob" that can be dragged along a track. Sliders can be horizontally or vertically oriented, as shown in Figure 24-30.

A slider is designed to represent a range of numeric values. At one end of the slider is the range's minimum value and at the other end is the range's maximum value. Both of the sliders shown in Figure 24-30 represent a range of 0 through 50. Sliders hold a numeric value in a field, and as the user moves the knob along the track, the numeric value is adjusted accordingly. Notice that the sliders in Figure 24-30 have accompanying tick marks. At every tenth value, a major tick mark is displayed along with a label indicating the value at that tick mark. Between the major tick marks are minor tick marks, which in this example are displayed at every second value. The appearance of tick marks, their spacing, and the appearance of labels can be controlled through methods in the `JSlider` class. The `JSlider` constructor has the following general format:

```
JSlider(int orientation, int minValue,
        int maxValue, int initialValue)
```

**Figure 24-30** A horizontal and a vertical slider (Oracle Corporate Counsel)



The first argument is an `int` specifying the slider's orientation. You should use one of the constants `JSlider.HORIZONTAL` or `JSlider.VERTICAL`. The second argument is the minimum value of the slider's range and the third argument is the maximum value of the slider's range. The fourth argument is the initial value of the slider, which determines the initial position of the slider's knob. For example, the following code could be used to create the sliders shown in Figure 24-30:

```
JSlider slider1 = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
JSlider slider2 = new JSlider(JSlider.VERTICAL, 0, 50, 25);
```

You set the major and minor tick mark spacing with the methods `setMajorTickSpacing` and `setMinorTickSpacing`. Each of these methods accepts an `int` argument that specifies the intervals of the tick marks. For example, the following code sets the `slider1` object's major tick mark spacing at 10, and its minor tick mark spacing at 2:

```
slider1.setMajorTickSpacing(10);
slider1.setMinorTickSpacing(2);
```

If the `slider1` component's range is 0 through 50, then these statements would cause major tick marks to be displayed at values 0, 10, 20, 30, 40, and 50. Minor tick marks would be displayed at values 2, 4, 6, and 8, then at values 12, 14, 16, and 18, and so forth.

By default, tick marks are not displayed, and setting their spacing does not cause them to be displayed. You display tick marks by calling the `setPaintTicks` method, which accepts a `boolean` argument. If you pass `true`, then tick marks are displayed. If you pass `false`, they are not displayed. Here is an example:

```
slider1.setPaintTicks(true);
```

By default, labels are not displayed either. You display numeric labels on the slider component by calling the `setPaintLabels` method, which accepts a `boolean` argument. If you pass `true`, then numeric labels are displayed at the major tick marks. If you pass `false`, labels are not displayed. Here is an example:

```
slider1.setPaintLabels(true);
```

When the knob's position is moved, the slider component generates a change event. To handle the change event, you must write a change listener class. When you write a change listener class, it must meet the following requirements:

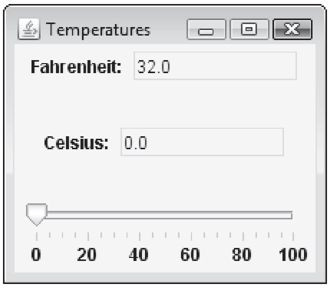
- It must implement the `ChangeListener` interface. This interface is in the `javax.swing.event` package.
- It must have a method named `stateChanged`. This method must take an argument of the `ChangeEvent` type.

To retrieve the current value stored in a `JSlider`, use the `getValue` method. This method returns the slider's value as an `int`. Here is an example:

```
currentValue = slider1.getValue();
```

The `TempConverter` class shown in Code Listing 24-7 demonstrates the `JSlider` component. This class displays the window shown in Figure 24-31. Two temperatures are initially shown: 32.0 degrees Fahrenheit and 0.0 degrees Celsius. A slider, which has the range of 0 through 100, allows you to adjust the Celsius temperature and immediately see the Fahrenheit conversion. The `main` method creates an instance of the class and displays the window.

**Figure 24-31** Window displayed by the `TempConverterWindow` class (Oracle Corporate Counsel)



Temperatures

Fahrenheit: 32.0

Celsius: 0.0

**Code Listing 24-7** (`TempConverter.java`)

```java
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

/**
   This class displays a window with a slider component.
   The user can convert the Celsius temperatures from
   0 through 100 to Fahrenheit by moving the slider.
*/

public class TempConverter extends JFrame
{
   private JLabel label1, label2;     // Message labels
   private JTextField fahrenheitTemp; // Fahrenheit temp
   private JTextField celsiusTemp;    // Celsius temp
   private JPanel fpanel;             // Fahrenheit panel
   private JPanel cpanel;             // Celsius panel
   private JPanel sliderPanel;        // Slider panel
   private JSlider slider;            // Temperature adjuster

   /**
      Constructor
   */

   public TempConverter()
   {
      // Set the title.
      setTitle("Temperatures");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the message labels.
      label1 = new JLabel("Fahrenheit: ");
      label2 = new JLabel("Celsius: ");

      // Create the read-only text fields.
      fahrenheitTemp = new JTextField("32.0", 10);
      fahrenheitTemp.setEditable(false);
      celsiusTemp = new JTextField("0.0", 10);
      celsiusTemp.setEditable(false);

      // Create the slider.
      slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 0);
      slider.setMajorTickSpacing(20); // Major tick every 20
      slider.setMinorTickSpacing(5);  // Minor tick every 5
      slider.setPaintTicks(true);     // Display tick marks
      slider.setPaintLabels(true);    // Display numbers
      slider.addChangeListener(new SliderListener());

      // Create panels and place the components in them.
      fpanel = new JPanel();
      fpanel.add(label1);
      fpanel.add(fahrenheitTemp);
      cpanel = new JPanel();
      cpanel.add(label2);
      cpanel.add(celsiusTemp);
      sliderPanel = new JPanel();
      sliderPanel.add(slider);

      // Create a GridLayout manager.
      setLayout(new GridLayout(3, 1));

      // Add the panels to the content pane.
      add(fpanel);
      add(cpanel);
      add(sliderPanel);

      // Pack and display the frame.
      pack();
      setVisible(true);
   }

   /**
      Private inner class to handle the change events
      that are generated when the slider is moved.
   */

   private class SliderListener implements ChangeListener
   {
      public void stateChanged(ChangeEvent e)
      {
         double fahrenheit, celsius;

         // Get the slider value.
         celsius = slider.getValue();

         // Convert the value to Fahrenheit.
         fahrenheit = (9.0 / 5.0) * celsius + 32.0;

         // Store the celsius temp in its display field.
         celsiusTemp.setText(Double.toString(celsius));

         // Store the Fahrenheit temp in its display field.
         fahrenheitTemp.setText(String.format("%.1f", fahrenheit));
      }
   }

   /*
      The main method creates an instance of the
      class, which displays a window with a slider.
   */

   public static void main(String[] args)
   {
      new TempConverter();
   }
}
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

24.29 What type of event does a `JSlider` generate when its slider knob is moved?

24.30 What `JSlider` methods do you use to perform each of these operations?

a) Establish the spacing of major tick marks.
b) Establish the spacing of minor tick marks.
c) Cause tick marks to be displayed.
d) Cause labels to be displayed.

## 24.11 Look and Feel

**CONCEPT: A GUI application's appearance is determined by its look and feel. Java allows you to select an application's look and feel.**

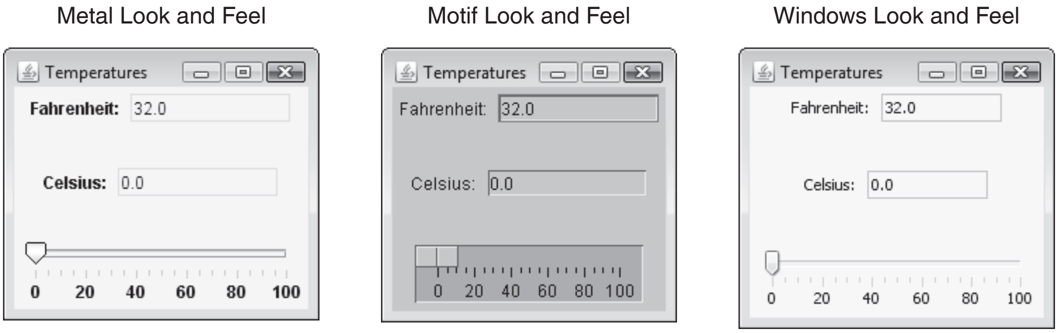
Most operating systems' GUIs have their own unique appearance and style conventions. For example, if a Windows user switches to a Macintosh, UNIX, or Linux system, the first thing he or she is likely to notice is the difference in the way the GUIs on each system appear. The appearance of a particular system's GUI is known as its look and feel.

Java allows you to select the look and feel of a GUI application. The default look and feel for Java is called *Ocean*. This is the look and feel that you have seen in all of the GUI applications that we have written in this text. Some of the other look and feel choices are Metal, Motif, and Windows. Metal was the default look and feel for previous versions of Java. Motif is similar to a UNIX look and feel. Windows is the look and feel of the Windows operating system. Figure 24-32 shows how the `TempConverterWindow` class window, presented earlier in this chapter, appears in each of these looks and feels.

**NOTE:** Ocean is actually a special theme of the Metal look and feel.

**NOTE:** Currently the Windows look and feel is available only on computers running the Microsoft Windows operating system.

**Figure 24-32** Metal, Motif, and Windows looks and feels (Oracle Corporate Counsel)



Metal Look and Feel

Fahrenheit: 32.0

Celsius: 0.0

Motif Look and Feel

Fahrenheit: 32.0

Celsius: 0.0

Windows Look and Feel

Fahrenheit: 32.0

Celsius: 0.0

To change an application's look and feel, you call the `UIManager` class's static `setLookAndFeel` method. Java has a class for each look and feel, and this method takes the fully qualified class name for the desired look and feel as its argument. The class name must be passed as a string. Table 24-1 lists the fully qualified class names for the Metal, Motif, and Windows looks and feels.

**Table 24-1** Look and feel class names

| Class Name | Look and Feel |
|---|---|
| `"javax.swing.plaf.metal.MetalLookAndFeel"` | Metal |
| `"com.sun.java.swing.plaf.motif.MotifLookAndFeel"` | Motif |
| `"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"` | Windows |

When you call the `UIManager.setLookAndFeel` method, any components that have already been created need to be updated. You do this by calling the `SwingUtilities.updateComponentTreeUI` method, passing a reference to the component that you want to update as an argument.

The `UIManager.setLookAndFeel` method throws a number of exceptions. Specifically, it throws `ClassNotFoundException`, `InstantiationException`, `IllegalAccessException`, and `UnsupportedLookAndFeelException`. Unless you want to trap each of these types of exceptions, you can simply trap exceptions of type `Exception`. Here is an example of code that can be run from a `JFrame` object that changes its look and feel to Motif:

```
try
{
    UIManager.setLookAndFeel(
         "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
    JOptionPane.showMessageDialog(null, "Error setting " +
                                  "the look and feel.");
    System.exit(0);
}
```

And here is an example of code that can be run from a `JFrame` object that changes its look and feel to Windows:

```
try
{
    UIManager.setLookAndFeel(
         "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
    JOptionPane.showMessageDialog(null, "Error setting " +
                                  "the look and feel.");
    System.exit(0);
}
```

## 24.12 Common Errors to Avoid

- **Only retrieving the first selected item from a list component in which multiple items have been selected.** If multiple items have been selected in a list component, the `getSelectedValue` method returns only the first selected item. Likewise, the `getSelectedIndex` method returns only the index of the first selected item. You should use the `getSelectedValues` or `getSelectedIndices` methods instead.
- **Using 1 as the beginning index for a list or combo box.** The indices for a list or combo box start at 0, not 1.
- **Forgetting to add a list or text area to a scroll pane.** The `JList` and `JTextArea` components do not automatically display scroll bars. You must add these components to a scroll pane object in order for them to display scroll bars.
- **Using the add method instead of the constructor to add a component to a scroll pane.** To add a component to a scroll pane, you must pass a reference to the component as an argument to the `JScrollPane` constructor.
- **Adding a component to a scroll pane and then adding the component (not the scroll pane) to another container, such as a panel.** If you add a component to a scroll pane and then intend to add that same component to a panel or other container, you must add the scroll pane instead of the component. Otherwise, the scroll bars will not appear on the component.

- **Forgetting to call the `setEditable` method to give a combo box a text field.** By default, a combo box is the combination of a button and a list. To make it a combination of a text field and a list, you must call the `setEditable` method and pass `true` as an argument.
- **Trying to open an image file of an unsupported type.** Currently, an `ImageIcon` object can open image files that are stored in JPEG, GIF, or PNG formats.
- **Loading an image into an existing `JLabel` component and clipping part of the image.** If you have not explicitly set the preferred size of a `JLabel` component, it resizes itself automatically when you load an image into it. The `JFrame` that encloses the `JLabel` does not automatically resize, however. You must call the `JFrame` object's `pack` method or `setPreferredSize` method to resize it.
- **Assigning the same mnemonic to more than one component.** If you assign the same mnemonic to more than one component in a window, it works only for the first component that you assigned it to.
- **Forgetting to add menu items to a `JMenu` component, and `JMenu` components to a `JMenuBar` component.** After you create a menu item, you must add it to a `JMenu` component in order for it to be displayed on the menu. Likewise, `JMenu` components must be added to a `JMenuBar` component in order to be displayed on the menu bar.
- **Not calling the `JFrame` object's `setJMenuBar` method to place the menu bar.** To display a menu bar, you must call the `setJMenuBar` method and pass it as an argument.
- **Not grouping `JRadioButtonMenuItems` in a `ButtonGroup` object.** Just like regular radio button components, you must group radio button menu items in a button group in order to create a mutually exclusive relationship among them.

## Review Questions and Exercises

### Multiple Choice and True/False

1. You can use this method to make a text field read-only.
   a. `setReadOnly`
   b. `setChangeable`
   c. `setUneditable`
   d. `setEditable`
2. A `JList` component generates this type of event when the user selects an item.
   a. action event
   b. item event
   c. list selection event
   d. list change event
3. To display a scroll bar with a `JList` component, you must ________.
   a. do nothing; the `JList` automatically appears with scroll bars if necessary
   b. add the `JList` component to a `JScrollPane` component
   c. call the `setScrollBar` method
   d. none of the above; you cannot display a scroll bar with a `JList` component

4. This is the `JList` component's default selection mode.
   a. single selection
   b. single interval selection
   c. multiple selection
   d. multiple interval selection
5. A list selection listener must have this method.
   a. `valueChanged`
   b. `selectionChanged`
   c. `actionPerformed`
   d. `itemSelected`
6. The `ListSelectionListener` interface is in this package.
   a. `java.awt`
   b. `java.awt.event`
   c. `javax.swing.event`
   d. `javax.event`
7. This `JList` method returns −1 if no item in the list is selected.
   a. `getSelectedValue`
   b. `getSelectedItem`
   c. `getSelectedIndex`
   d. `getSelection`
8. A `JComboBox` component generates this type of event when the user selects an item.
   a. action event
   b. item event
   c. list selection event
   d. list change event
9. You can pass an instance of this class to the `JLabel` constructor if you want to display an image in the label.
   a. `ImageFile`
   b. `ImageIcon`
   c. `JLabelImage`
   d. `JImageFile`
10. This method can be used to store an image in a `JLabel` or a `JButton` component.
    a. `setImage`
    b. `storeImage`
    c. `getIcon`
    d. `setIcon`
11. This is text that appears in a small box when the user holds the mouse cursor over a component.
    a. mnemonic
    b. instant message
    c. tool tip
    d. pop-up mnemonic

12. This is a key that activates a component just as if the user clicked it with the mouse.
    a. mnemonic
    b. key activator
    c. tool tip
    d. click simulator
13. To display an open file or save file dialog box, you use this class.
    a. `JFileChooser`
    b. `JOpenSaveDialog`
    c. `JFileDialog`
    d. `JFileOptionPane`
14. To display a dialog box that allows the user to select a color, you use this class.
    a. `JColor`
    b. `JColorDialog`
    c. `JColorChooser`
    d. `JColorOptionPane`
15. You use this class to create a menu bar.
    a. `MenuBar`
    b. `JMenuBar`
    c. `JMenu`
    d. `JBar`
16. You use this class to create a radio button menu item.
    a. `JMenuItem`
    b. `JRadioButton`
    c. `JRadioButtonItem`
    d. `JRadioButtonMenuItem`
17. You use this method to place a menu bar on a `JFrame`.
    a. `setJMenuBar`
    b. `setMenuBar`
    c. `placeMenuBar`
    d. `setJMenu`
18. The `setPreferredSize` method accepts this as its argument(s).
    a. a `Size` object
    b. two `int` values
    c. a `Dimension` object
    d. one `int` value
19. Components of this class are multi-line text fields.
    a. `JMultiLineTextField`
    b. `JTextArea`
    c. `JTextField`
    d. `JEditField`

20. This method is inherited from `JComponent` and changes the appearance of a component's text.
    a. `setAppearance`
    b. `setTextAppearance`
    c. `setFont`
    d. `setText`
21. This method sets the intervals at which major tick marks are displayed on a `JSlider` component.
    a. `setMajorTickSpacing`
    b. `setMajorTickIntervals`
    c. `setTickSpacing`
    d. `setIntervals`
22. **True or False:** You can use code to change the contents of a read-only text field.
23. **True or False:** A `JList` component automatically appears with a line border drawn around it.
24. **True or False:** In single interval selection mode, the user may select multiple items from a `JList` component.
25. **True or False:** With an editable combo box the user may only enter a value that appears in the component's list.
26. **True or False:** You can store either text or an image in a `JLabel` object, but not both.
27. **True or False:** You can store large images as well as small ones in a `JLabel` component.
28. **True or False:** Mnemonics are useful for users who are good with the keyboard.
29. **True or False:** A `JMenuBar` object acts as a container for `JMenu` components.
30. **True or False:** A `JMenu` object cannot contain other `JMenu` objects.
31. **True or False:** A `JTextArea` component does not automatically display scroll bars.
32. **True or False:** By default, a `JTextArea` component does not perform line wrapping.
33. **True or False:** A `JSlider` component generates an action event when the slider knob is moved.
34. **True or False:** By default, a `JSlider` component displays labels and tick marks.
35. **True or False:** When labels are displayed on a `JSlider` component, they are displayed on the major tick marks.

## Find the Error

1.
```
// Create a read-only text field.
JTextField textField = new JTextField(10);
textField.setEditable(true);
```
2.
```
// Create a black 1-pixel border around list, a JList component.
list.setBorder(Color.BLACK, 1);
```
3.
```
// Create a JList and add it to a scroll pane.
// Assume that array already exists.
JList list = new JList(array);
JScrollPane scrollPane = new JScrollPane();
scrollPane.add(list);
```

4. 
   ```
   // Assume that nameBox is a combo box and is properly set up
   // with a list of names to choose from.
   // Get value of the selected item.
   String selectedName = nameBox.getSelectedIndex();
   ```
5. 
   ```
   JLabel label = new JLabel("Have a nice day!");
   label.setImage(image);
   ```
6. 
   ```
   // Add a menu to the menu bar.
   JMenuBar menuBar = new JMenuBar(menuItem);
   ```
7. 
   ```
   // Create a text area with 20 columns and 5 rows.
   JTextArea textArea = new JTextArea (20, 5);
   ```

### Algorithm Workbench

1. Give an example of code that creates a read-only text field.
2. Write code that creates a list with the following items: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.
3. Write code that adds a scroll bar to the list you created in your answer to Algorithm Workbench 2.
4. Assume that the variable `myList` references a `JList` component, and `selection` is a `String variable`. Write code that assigns the selected item in the `myList` component to the `selection` variable.
5. Assume that the variable `myComboBox` references an uneditable combo box, and `selectionIndex` is an `int` variable. Write code that assigns the index of the selected item in the `myComboBox` component to the `selectionIndex` variable.
6. Write code that stores the image in the file *dog.jpg* in a label.
7. Assume that `label` references an existing `JLabel` object. Write code that stores the image in the file *picture.gif* in the label.
8. Write code that creates a button with the text "Open File." Assign the O key as a mnemonic and assign "This button opens a file" as the component's tool tip.
9. Write code that displays a file open dialog box. If the user selects a file, the code should store the file's path and name in a `String` variable.
10. Write code that creates a text area displaying 10 rows and 15 columns. The text area should be capable of displaying scroll bars, when necessary. It should also perform word style line wrapping.
11. Write the code that creates a menu bar with one menu named File. The File menu should have the F key assigned as a mnemonic. The File menu should have three menu items: Open, Print, and Exit. Assign mnemonic keys of your choice to each of these items. Register an instance of the `OpenListener` class as an action listener for the Open menu item, an instance of the `PrintListener` class as an action listener for the Print menu item, and an instance of the `ExitListener` class as an action listener for the Exit menu item. Assume these classes have already been created.
12. Write code that creates a `JSlider` component. The component should be horizontally oriented and its range should be 0 through 1000. Labels and tick marks should be displayed. Major tick marks should appear at every 100th number, and minor tick marks should appear at every 25th number. The initial value of the slider should be set at 500.

### Short Answer

1. What selection mode should you select if you want the user to select a single item only in a list?
2. You want to provide 20 items in a list for the user to select from. Which component would take up less space, a `JList` or a `JComboBox`?
3. What is the difference between an uneditable combo box and an editable combo box? Which one is a combo box by default?
4. Describe how you can store both an image and text in a `JLabel` component.
5. What is a mnemonic? How does the user use it?
6. What happens when the mnemonic that you assign to a component is a letter that appears in the component's text?
7. What is a tool tip? What is its purpose?
8. What do you do to a group of radio button menu items so that only one of them can be selected at a time?
9. When a checked menu item shows a check mark next to it, what happens when the user clicks on it?
10. What fonts does Java guarantee you have?
11. Why would a `JSlider` component be ideal when you want the user to enter a number, but you want to make sure that the number is within a range?
12. What are the standard GUI looks and feels that are available in Java?

## Programming Challenges

MyProgrammingLab™

*Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. Scrollable Tax Calculator

Create an application that allows you to enter the amount of a purchase and then displays the amount of sales tax on that purchase. Use a slider to adjust the tax rate between 0 percent and 10 percent.

VideoNote

The Image Viewer Problem

### 2. Image Viewer

Write an application that allows the user to view image files. The application should use either a button or a menu item that displays a file chooser. When the user selects an image file, it should be loaded and displayed.

### 3. Dorm and Meal Plan Calculator

A university has the following dormitories:

Allen Hall: \$1,500 per semester
Pike Hall: \$1,600 per semester
Farthing Hall: \$1,200 per semester
University Suites: \$1,800 per semester

The university also offers the following meal plans:

7 meals per week: $560 per semester
14 meals per week: $1,095 per semester
Unlimited meals: $1,500 per semester

Create an application with two combo boxes. One should hold the names of the dormitories, and the other should hold the meal plans. The user should select a dormitory and a meal plan, and the application should show the total charges for the semester.

### 4. Skateboard Designer

The Skate Shop sells the skateboard products listed in Table 24-2.

**Table 24-2** Skateboard products

| Decks | Truck Assemblies | Wheels |
|---|---|---|
| The Master Thrasher $60 | 7.75 inch axle $35 | 51 mm $20 |
| The Dictator $45 | 8 inch axle $40 | 55 mm $22 |
| The Street King $50 | 8.5 inch axle $45 | 58 mm $24 |
| | | 61 mm $28 |

In addition, the Skate Shop sells the following miscellaneous products and services:

Grip tape: $10
Bearings: $30
Riser pads: $2
Nuts & bolts kit: $3

Create an application that allows the user to select one deck, one truck assembly, and one wheel set from either list components or combo boxes. The application should also have a list component that allows the user to select multiple miscellaneous products. The application should display the subtotal, the amount of sales tax (at 6 percent), and the total of the order.

### 5. Shopping Cart System

Create an application that works like a shopping cart system for a bookstore. In this chapter's source code folder (available on the book's companion Web site at www.pearson.com/gaddis), you will find a file named *BookPrices.txt*. This file contains the names and prices of various books, formatted in the following fashion:

I Did It Your Way, $11.95
The History of Scotland, $14.50
Learn Calculus in One Day, $29.95
Feel the Stress, $18.50

Each line in the file contains the name of a book, followed by a comma, followed by the book's retail price. When your application begins execution, it should read the contents of the file and store the book titles in a list component. The user should be able to select a title from the list and add it to a shopping cart, which is simply another list component. The application should have buttons or menu items that allow the user to remove items from the shopping cart, clear the shopping cart of all selections, and check out. When the user checks out, the application should calculate and display the subtotal of all the books in the shopping cart, the sales tax (which is 6 percent of the subtotal), and the total.

### 6. Cell Phone Packages

Cell Solutions, a cell phone provider, sells the following packages:

300 minutes per month: $45.00 per month
800 minutes per month: $65.00 per month
1500 minutes per month: $99.00 per month

The provider sells the following phones (a 6 percent sales tax applies to the sale of a phone):

Model 100: $29.95
Model 110: $49.95
Model 200: $99.95

Customers may also select the following options:

Voice mail: $5.00 per month
Text messaging: $10.00 per month

Write an application that displays a menu system. The menu system should allow the user to select one package, one phone, and any of the options desired. As the user selects items from the menu, the application should show the prices of the items selected.

### 7. Shade Designer

A custom window shade designer charges a base fee of $50 per shade. In addition, charges are added for certain styles, sizes, and colors as follows:

Styles:

Regular shades: Add $0
Folding shades: Add $10
Roman shades: Add $15

Sizes:

25 inches wide: Add $0
27 inches wide: Add $2
32 inches wide: Add $4
40 inches wide: Add $6

Colors:

Natural: Add $5
Blue: Add $0
Teal: Add $0
Red: Add $0
Green: Add $0

Create an application that allows the user to select the style, size, color, and number of shades from lists or combo boxes. The total charges should be displayed.

### 8. Conference Registration System

Create an application that calculates the registration fees for a conference. The general conference registration fee is $895 per person, and student registration is $495 per person. There is also an optional opening night dinner with a keynote speech for $30 per person. In addition, the optional preconference workshops listed in Table 24-3 are available.

**Table 24-3** Optional preconference workshops

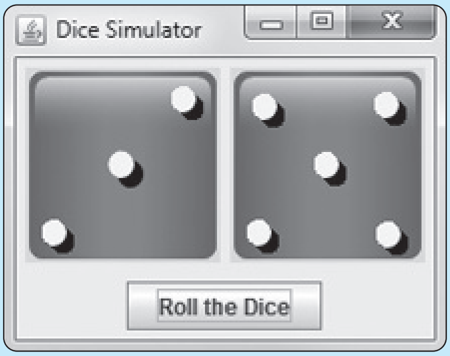
| Workshop | Fee |
|---|---|
| Introduction to E-commerce | $295 |
| The Future of the Web | $295 |
| Advanced Java Programming | $395 |
| Network Security | $395 |

The application should allow the user to select the registration type, the optional opening night dinner and keynote speech, and as many preconference workshops as desired. The total cost should be displayed.

### 9. Dice Simulator

Write a GUI application that simulates a pair of dice, similar to that shown in Figure 24-33. Each time the button is clicked, the application should roll the dice, using random numbers to determine the value of each die. (This chapter's source code folder contains images that you can use to display the dice.)

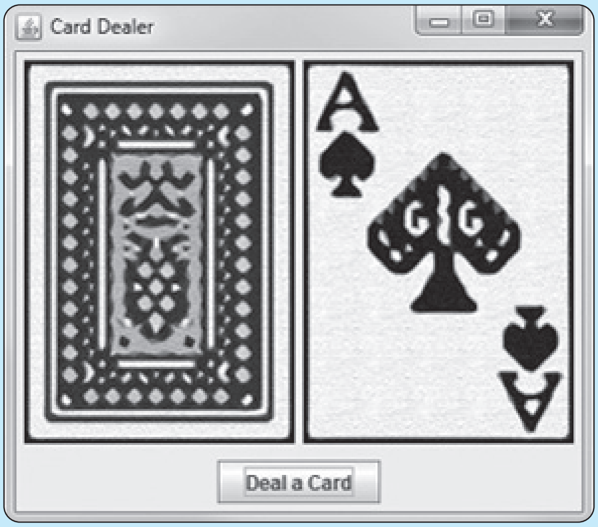
**Figure 24-33** Dice simulator (Oracle Corporate Counsel)



### 10. Card Dealer

This chapter's source code folder contains images for a complete deck of poker cards. Write a GUI application, similar to the one shown in Figure 24-34, that randomly selects a card from the deck and displays it each time the user clicks the button. When a card has been selected, it is removed from the deck and cannot be selected again. Display a message when no more cards are left in the deck.

**Figure 24-34** Card dealer (Oracle Corporate Counsel)

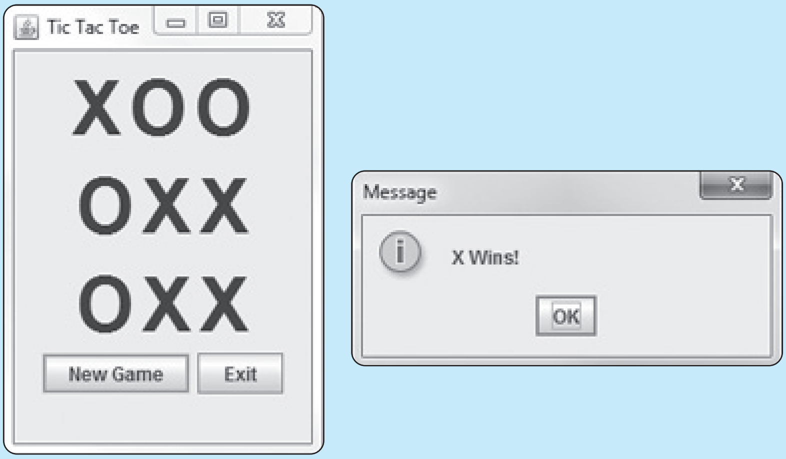


### 11. Tic Tac Toe Simulator

Create a GUI application that simulates a game of tic tac toe. Figure 24-35 shows an example of the application's window. The window shown in the figure uses nine large `JLabel` components to display the Xs and Os.

One approach in designing this application is to use a two-dimensional `int` array to simulate the game board in memory. When the user clicks the *New Game* button, the application should step through the array, storing a random number in the range of 0 through 1 in each element. The number 0 represents the letter O, and the number 1 represents the letter X. The `JLabel` components should then be updated to display the game board. The application should display a message indicating whether player X won, player Y won, or the game was a tie.

**Figure 24-35** The Tic Tac Toe application (Oracle Corporate Counsel)



XOO
OXX
OXX

X Wins!

# CHAPTER 25

# Applets and More

**TOPICS**

25.1 Introduction to Applets
25.2 A Brief Introduction to HTML
25.3 Creating Applets with Swing
25.4 Using AWT for Portability
25.5 Drawing Shapes
25.6 Handling Mouse Events
25.7 `Timer` Objects
25.8 Playing Audio
25.9 Common Errors to Avoid

**NOTE:** Java applets are an older technology, and Oracle has announced that beginning with Java 9, the browser plug-in needed to support Java applets has been discontinued. This is a legacy chapter from the book, and is provided for students and instructors making the transition from applets to other technologies. As you read this chapter, keep in mind that newer technologies exist as alternatives to applets.

## 25.1 Introduction to Applets

**CONCEPT: An applet is a Java program that is associated with a Web page and is executed in a Web browser as part of that Web page.**

Recall from Chapter 1 that there are two types of programs you can create with Java: applications and applets. An *application* is a stand-alone program that runs on your computer. So far in this text, we have concentrated exclusively on writing applications.

*Applets* are Java programs that are usually part of a Web site. If a user opens the Web site with a Java-enabled browser, the applet is executed inside the browser window. It appears to the user that the applet is part of the Web site. This is how it works: Applets are stored on a Web server along with the site's Web pages. When a user accesses a Web page on a server with his or her browser, any applets associated with the Web page are transmitted over the Internet from the server to the user's system. This is illustrated in Figure 25-1. Once the applets are transmitted, the user's system executes them.

Applets are important because they can be used to extend the capabilities of a Web page. Web pages are normally written in Hypertext Markup Language (HTML). HTML is limited, however, because it merely describes the content and layout of a Web page, and creates links to other files and Web pages. HTML does not have sophisticated abilities such as performing math calculations and interacting with the user. A programmer can write a Java applet to perform these types of operations and associate it with a Web page. When someone visits the Web page, the applet is downloaded to the visitor's browser and executed.

**Figure 25-1** Applets are transmitted along with Web pages

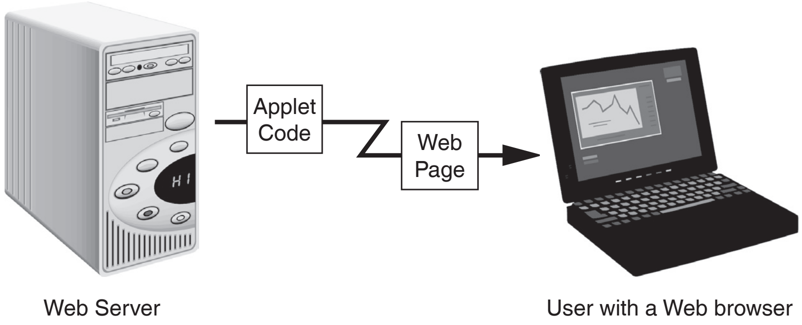
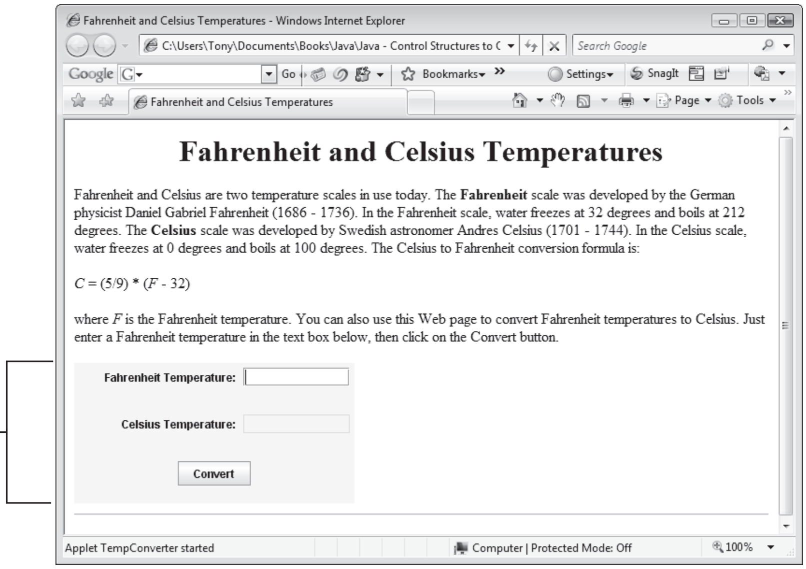

Applet Code
Web Page
Web Server
User with a Web browser

Figure 25-2 shows an example of a Web page that has an applet. In the figure, the Web page is being viewed with Internet Explorer. This Web page briefly explains the Fahrenheit and Celsius temperature scales. The area with the text boxes and the button at the bottom of the page is generated by an applet. To see a Fahrenheit temperature converted to Celsius, the user can enter the Fahrenheit temperature into the top text box and click the Convert button. The Celsius temperature will be displayed in the read-only text box.

> An applet does not have to be on a Web server to be executed. The Web page shown in Figure 25-2 is in the source code folder *Chapter 19\TempConverter*. Open the *TempConverter.html* file in your Web browser to try it. Later in this chapter, we will take a closer look at this Web page and its applet.

**Figure 25-2** A Web page with an applet (Microsoft Corporation)


Fahrenheit and Celsius Temperatures - Windows Internet Explorer
C:\Users\Tony\Documents\Books\Java\Java - Control Structures to C
Search Google
Google
Go
Bookmarks
Settings
SnagIt
Fahrenheit and Celsius Temperatures
Page
Tools
Computer | Protected Mode: Off
100%

This part of the Web page is generated by an applet.

**Fahrenheit and Celsius Temperatures**

Fahrenheit and Celsius are two temperature scales in use today. The **Fahrenheit** scale was developed by the German physicist Daniel Gabriel Fahrenheit (1686 - 1736). In the Fahrenheit scale, water freezes at 32 degrees and boils at 212 degrees. The **Celsius** scale was developed by Swedish astronomer Andres Celsius (1701 - 1744). In the Celsius scale, water freezes at 0 degrees and boils at 100 degrees. The Celsius to Fahrenheit conversion formula is:

*C* = (5/9) * (*F* - 32)

where *F* is the Fahrenheit temperature. You can also use this Web page to convert Fahrenheit temperatures to Celsius. Just enter a Fahrenheit temperature in the text box below, then click on the Convert button.

**Fahrenheit Temperature:**

**Celsius Temperature:**

**Convert**

Applet TempConverter started

Most Web browsers have a special version of the JVM for running applets. For security purposes, this version of the JVM greatly restricts what an applet can do. Here is a summary of the restrictions placed on applets:

- Applets cannot delete files, read the contents of files, or create files on the user's system.
- Applets cannot run any other program on the user's system.
- Applets cannot execute operating system procedures on the user's system.
- Applets cannot retrieve information about the user's system, or the user's identity.
- Applets cannot make network connections with any system except the server from which the applet was transmitted.
- If an applet displays a window, it will automatically have a message such as "Warning: Applet Window" displayed in it. This lets the user know that the window was not displayed by an application on his or her system.

These restrictions might seem severe, but they are necessary to prevent malicious code from attacking or spying on unsuspecting users. If an applet attempts to violate one of these restrictions, an exception is thrown.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

25.1 How is an applet that is associated with a Web page executed on a user's system?

25.2 Why do applets run in a restricted environment?

## 25.2 A Brief Introduction to HTML

**CONCEPT: When creating a Web page, you use Hypertext Markup Language (HTML) to create a file that can be read and processed by a Web browser.**

Hypertext Markup Language (HTML) is the language that Web pages are written in. Although it is beyond the scope of this text to teach you everything about HTML, this section will give you enough of the fundamentals so that you can write simple Web pages. You will need to know a little about HTML in order to run Java applets. If you are already familiar with HTML, this section is optional.

Before we continue, let's look at the meanings of the terms *hypertext* and *markup language*.

### Hypertext

Web pages can contain regular text and hypertext, which are both displayed in the browser window. In addition, *hypertext* can contain a link to another Web page, or perhaps another location in the same Web page. When the user clicks on the hypertext, it loads the Web page or the location that the hypertext is linked to.

## Markup Language

Although HTML is called a language, it is not a programming language like Java. Instead, HTML is a *markup language*. It allows you to "mark up" a text file by inserting special instructions. These instructions tell the browser how to format the text and create any hypertext links.

To make a Web page, you create a text file that contains HTML instructions, which are known as *tags*, as well as the text that should be displayed on the Web page. The resulting file is known as an *HTML document*, and it is usually saved with the *.html* file name extension. When a Web browser reads the HTML document, the tags instruct it how to format the text, where to place images, what to do when the user clicks on a link, and more.

Most HTML tags come in pairs. The first is known as the opening tag and the second is known as the closing tag. The general format of a simple tag is as follows:

```
<tag_name>
Text
</tag_name>
```

In this general format, `tag_name` is the name of the tag. The opening tag is `<tag_name>` and the closing tag is `</tag_name>`. Both the opening and closing tags are enclosed in angle brackets (`< >`). Notice that in the closing tag, the tag name is preceded by a forward slash (`/`). The `Text` that appears between the opening and closing tags is text that is formatted or modified by the tags.

## Document Structure Tags

Some of the HTML tags are used to establish the structure of an HTML document. The first of the structure tags that you should learn is the `<html></html>` tag. This tag marks the beginning and ending of an HTML document. Everything that appears between these tags, including other tags, is the content of the Web page. When you are writing an HTML document, place an `<html>` tag at the very beginning, and an `</html>` tag at the very end.

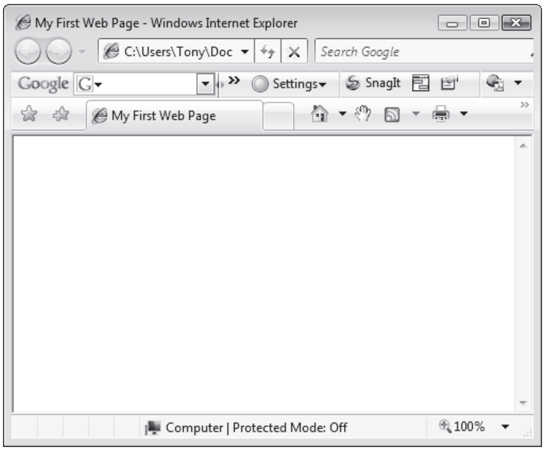
The next tag is `<head></head>`. Everything that appears between `<head>` and `</head>` is considered part of the document head. The *document head* is a section of the HTML file that contains information about the document. For example, key words that search engines use to identify a document are often placed in the document's head. The only thing that we will use the document head for is to display a title in the Web browser's title bar. You do this with the `<title></title>` tag. Any text that you place between `<title>` and `</title>` becomes the title of the page and is displayed in the browser's title bar. Code Listing 25-1 shows the contents of an HTML document with the title "My First Web Page".

Notice that the `<title></title>` tag is inside of the `<head></head>` tag. The only output displayed by this Web page is the title. Figure 25-3 shows how this Web page appears when opened in a browser.

**Code Listing 25-1** (`BasicWebPage1.html`)

```
<html>
<head>
    <title>My First Web Page</title>
</head>
</html>
```

**Figure 25-3** Web page with a title only (Microsoft Corporation)

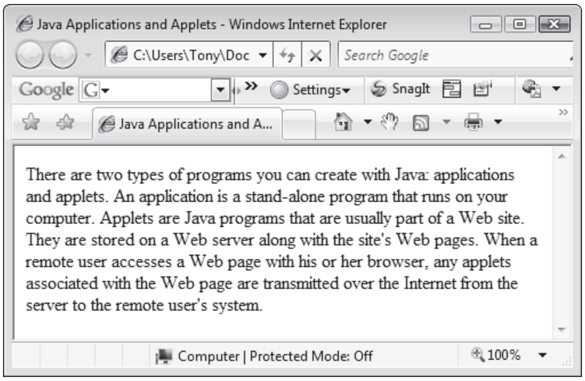

My First Web Page - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Settings
SnagIt
My First Web Page
Computer | Protected Mode: Off
100%

After the document head comes the document body, which is enclosed in the `<body></body>` tag. The *document body* contains all of the tags and text that produce output in the browser window. Code Listing 25-2 shows an HTML document with text placed in its body. Figure 25-4 shows the document when opened in a browser.

**Code Listing 25-2** (`BasicWebPage2.html`)

```
<html>
<head>
    <title>Java Applications and Applets</title>
</head>
<body>
    There are two types of programs you can create with Java: applications
    and applets. An application is a stand-alone program that runs on your
    computer. Applets are Java programs that are usually part of a Web site.
    They are stored on a Web server along with the site's Web pages. When a
    remote user accesses a Web page with his or her browser, any applets
    associated with the Web page are transmitted over the Internet from the
    server to the remote user's system.
</body>
</html>
```

**Figure 25-4** Web page produced by *BasicWebPage2.html* (Microsoft Corporation)



There are two types of programs you can create with Java: applications and applets. An application is a stand-alone program that runs on your computer. Applets are Java programs that are usually part of a Web site. They are stored on a Web server along with the site's Web pages. When a remote user accesses a Web page with his or her browser, any applets associated with the Web page are transmitted over the Internet from the server to the remote user's system.

Java Applications and Applets - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Settings
SnagIt
Java Applications and A...
Computer | Protected Mode: Off
100%

## Text Formatting Tags

The text displayed in the Web page in Figure 25-4 is unformatted, which means it appears as plain text. There are many HTML tags that you can use to change the appearance of text. For example, there are six different header tags that you can use to format text as a heading of some type. The `<h1></h1>` tag creates a level one header. A level one header appears in boldface, and is much larger than regular text. The `<h2></h2>` tag creates a level two header. A level two header also appears in boldface, but is smaller than a level one header. This pattern continues with the `<h3></h3>`, `<h4></h4>`, `<h5></h5>`, and `<h6></h6>` tags. The higher a header tag's level number is, the smaller the text that it formats appears. For example, look at the following HTML:

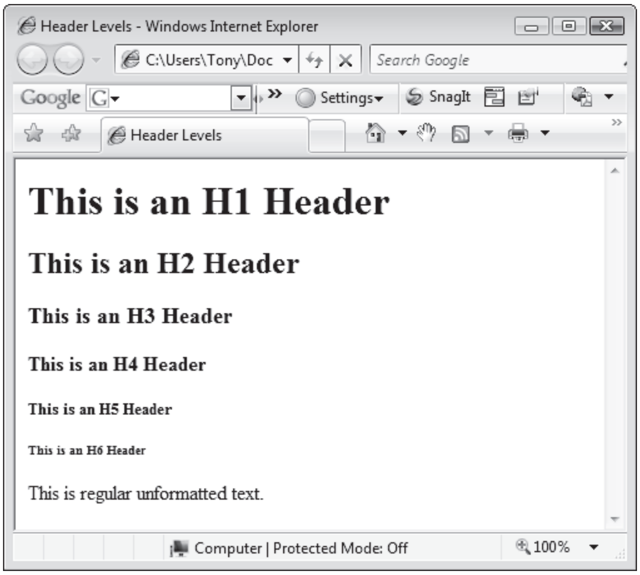
```
<h1>This is an h1 Header</h1>
<h2>This is an h2 Header</h2>
<h3>This is an h3 Header</h3>
<h4>This is an h4 Header</h4>
<h5>This is an h5 Header</h5>
<h6>This is an h6 Header</h6>
This is regular unformatted text.
```

When this appears in the body of an HTML document, it produces the Web page shown in Figure 25-5.

You can use the `<center></center>` tag to center a line of text in the browser window. To demonstrate, we will add the following line to the document that was previously shown in Code Listing 25-2:

```
<center><h1>Java</h1></center>
```

**Figure 25-5** Header levels (Microsoft Corporation)


Header Levels - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Settings
SnagIt
Header Levels
Computer | Protected Mode: Off
100%

# This is an H1 Header

## This is an H2 Header

### This is an H3 Header

#### This is an H4 Header

##### This is an H5 Header

###### This is an H6 Header

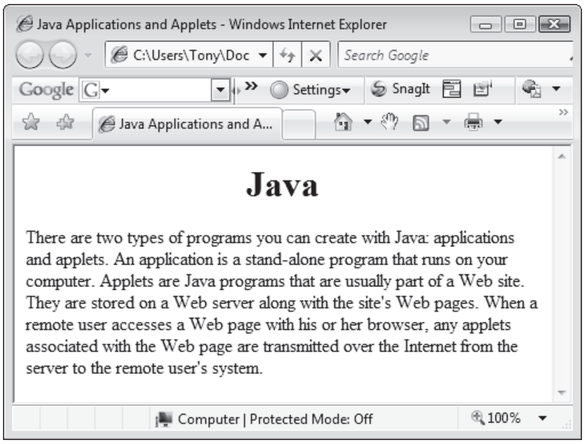
This is regular unformatted text.

This will cause the word "Java" to appear centered and as a level one header. The modified document is shown in Code Listing 25-3, and the Web page it produces is shown in Figure 25-6.

**Code Listing 25-3** (`BasicWebPage3.html`)

```
<html>
<head>
    <title>Java Applications and Applets</title>
</head>
<body>
    <center>
        <h1>Java</h1>
    </center>
    There are two types of programs you can create with Java: applications
    and applets. An application is a stand-alone program that runs
    on your computer. Applets are Java programs that are usually
    part of a Web site. They are stored on a Web server along with
    the site's Web pages. When a remote user accesses a Web page
    with his or her browser, any applets associated with the Web
    page are transmitted over the Internet from the server to the
    remote user's system.
</body>
</html>
```

**Figure 25-6** Web page produced by *BasicWebPage3.html* (Microsoft Corporation)


Java Applications and Applets - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Settings
SnagIt
Java Applications and A...
Computer | Protected Mode: Off
100%

Java

There are two types of programs you can create with Java: applications and applets. An application is a stand-alone program that runs on your computer. Applets are Java programs that are usually part of a Web site. They are stored on a Web server along with the site's Web pages. When a remote user accesses a Web page with his or her browser, any applets associated with the Web page are transmitted over the Internet from the server to the remote user's system.

Notice that in the HTML document, the word "Java" is enclosed in two sets of tags: the `<center>` tags and the `<h1>` tags. It doesn't matter which set of tags is used first. If we had written the line as follows, we would have gotten the same result:

```
<h1><center>Java</center></h1>
```

You can display text in boldface by using the `<b></b>` tag, and in italics by using the `<i></i>` tag. For example, the following will cause the text "Hello World" to be displayed in boldface:

```
<b>Hello World</b>
```

The following will cause "Hello World" to be displayed in italics:

```
<i>Hello World</i>
```

The following will display "Hello World" in boldface and italics:

```
<b><i>Hello World</i></b>
```

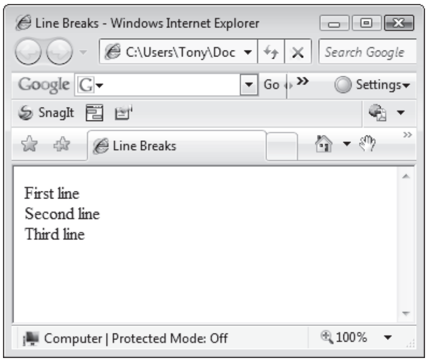
## Creating Breaks in Text

We will look at three HTML tags that are used to create breaks in a document's text. These three tags are unique from the ones we previously studied because they do not occur in pairs. When you use one of these tags, you only insert an opening tag.

The `<br />` tag causes a line break to appear at the point in the text where it is inserted. It is often necessary to insert `<br />` tags in an HTML document because the browser usually ignores the newline characters that are created when you press the Enter key. For example, if the following line appears in the body of an HTML document, it will cause the output shown in Figure 25-7.

```
First line<br />Second line<br />Third line
```

**Figure 25-7** Line breaks in an HTML document (Microsoft Corporation)

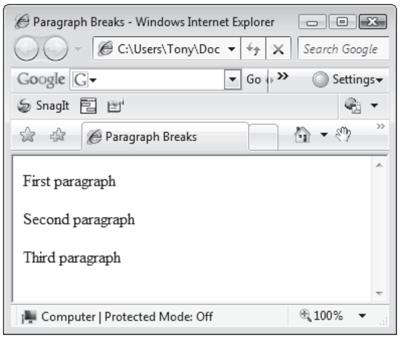

Line Breaks - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Go
Settings
SnagIt
Line Breaks
Computer | Protected Mode: Off
100%

First line
Second line
Third line

The `<p />` tag causes a paragraph break to appear at the point in the text where it is inserted. A paragraph break typically inserts more space into the text than a line break. For example, if the following line appears in the body of an HTML document, it will cause the output shown in Figure 25-8.

```
First paragraph<p />Second paragraph<p />Third paragraph
```

**Figure 25-8** Paragraph breaks in an HTML document (Microsoft Corporation)


Paragraph Breaks - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Go
Settings
SnagIt
Paragraph Breaks
Computer | Protected Mode: Off
100%

First paragraph

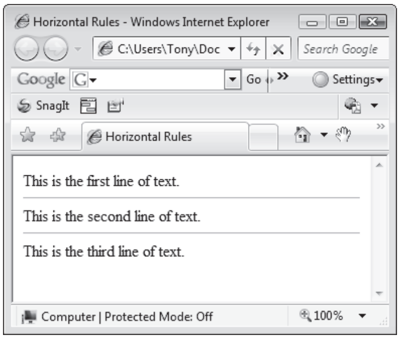
Second paragraph

Third paragraph

The `<hr />` tag causes a horizontal rule to appear at the point in the text where it is inserted. A horizontal rule is a thin, horizontal line that is drawn across the Web page. For example, if the following text appears in the body of an HTML document, it will cause the output shown in Figure 25-9.

```
This is the first line of text.
<hr />
This is the second line of text.
<hr />
This is the third line of text.
```

**Figure 25-9** Horizontal rules in a Web page (Microsoft Corporation)


Horizontal Rules - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Go
Settings
SnagIt
Horizontal Rules
Computer | Protected Mode: Off
100%

This is the first line of text.

This is the second line of text.

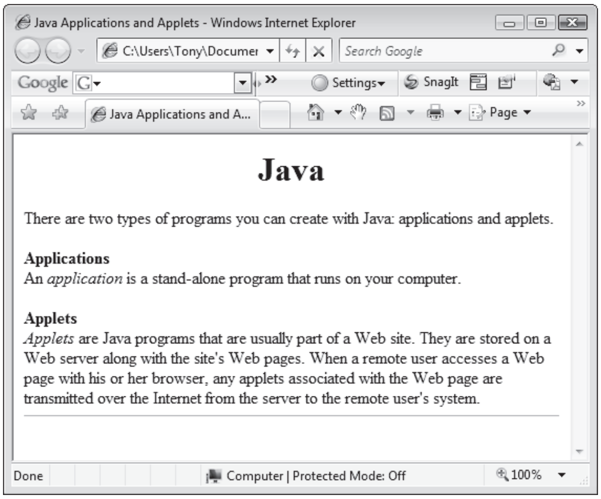
This is the third line of text.

The HTML document shown in Code Listing 25-4 demonstrates each of the tags we have discussed. The Web page it produces is shown in Figure 25-10.

**Code Listing 25-4** (BasicWebPage4.html)

```
<html>
<head>
    <title>Java Applications and Applets</title>
</head>
<body>
    <center>
        <h1>Java</h1>
    </center>
    There are two types of programs you can create with Java: applications
    and applets.
    <p />
    <b>Applications</b>
    <br />
    An <i>application</i> is a stand-alone program that runs on
    your computer.
    <p />
    <b>Applets</b>
    <br />
    <i>Applets</i> are Java programs that are usually part of a
    Web site. They are stored on a Web server along with the site's
    Web pages. When a remote user accesses a Web page with his or
    her browser, any applets associated with the Web page are
    transmitted over the Internet from the server to the remote
    user's system.
    <hr />
</body>
</html>
```

**Figure 25-10** Web page produced by *BasicWebPage4.html* (Microsoft Corporation)


Java Applications and Applets - Windows Internet Explorer
C:\Users\Tony\Documer
Search Google
Google
Settings
SnagIt
Java Applications and A...
Page
Done
Computer | Protected Mode: Off
100%

**Java**

There are two types of programs you can create with Java: applications and applets.

**Applications**
An *application* is a stand-alone program that runs on your computer.

**Applets**
*Applets* are Java programs that are usually part of a Web site. They are stored on a Web server along with the site's Web pages. When a remote user accesses a Web page with his or her browser, any applets associated with the Web page are transmitted over the Internet from the server to the remote user's system.

## Inserting Links

As previously mentioned, a link is some element in a Web page that can be clicked on by the user. When the user clicks the link, another Web page is displayed, or some sort of action is initiated. We now look at how to insert a simple link that causes another Web page to be displayed. The tag that is used to insert a link has the following general format:

```
<a href="Address">Text</a>
```

The `Text` that appears between the opening and closing tags is the text that will be displayed in the Web page. When the user clicks on this text, the Web page that is located at `Address` will be displayed in the browser. This address is often referred to as a *uniform resource locator* (*URL*). Notice that the address is enclosed in quotation marks. Here is an example:

```
<a href="http://www.pearsonhighered.com/gaddis/">Click here to go to the
textbook's web site.</a>
```

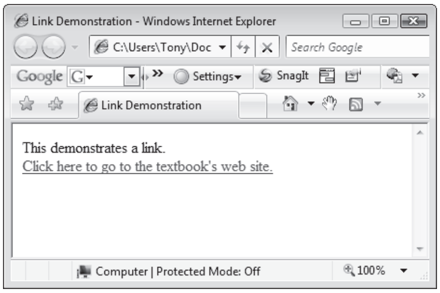
The HTML document shown in Code Listing 25-5 uses this link, and Figure 25-11 shows how the page appears in the browser.

**Code Listing 25-5** (`LinkDemo.html`)

```
<html>
<head>
    <title>Link Demonstration</title>
</head>
<body>
    This demonstrates a link.
    <br />
    <a href="http://www.aw.com/gaddis">Click here to go to the textbook's
    web site.</a>
</body>
</html>
```

The text that is displayed by a link is usually highlighted in some way to let the user know that it is not ordinary text. In Figure 25-11, the link text is underlined. When the user clicks on this text, the browser displays the Web page at `www.aw.com/gaddis`

**Figure 25-11** Web page produced by *LinkDemo.html* (Microsoft Corporation)


Link Demonstration - Windows Internet Explorer
C:\Users\Tony\Doc
Search Google
Google
Settings
SnagIt
Link Demonstration
Computer | Protected Mode: Off
100%

This demonstrates a link.
Click here to go to the textbook's web site.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

25.3 What tag marks the beginning and end of an HTML document?

25.4 What tag marks the beginning and end of an HTML document's head section?

25.5 What statement would you use in an HTML document to display the text "My Web Page" in the browser's title bar? What section of the HTML document would this statement be written in?

25.6 What tag marks the beginning and end of an HTML document's body section?

25.7 What statement would you write in an HTML document to display the text "Student Roster" as a level one header?

25.8 What statement would you write in an HTML document to display the text "My Resume" in bold and centered on the page?

25.9 What statement would you write in an HTML document to display the text "Hello World" in bold and italic?

25.10 What tag causes a line break? What tag causes a paragraph break? What tag displays a horizontal rule?

25.11 Suppose you wanted to display the text "Click Here" as a link to the Web site `http://java.sun.com`. What statement would you write to create the text?

## 25.3 Creating Applets with Swing

**CONCEPT: You extend a class from `JApplet` to create an applet, just as you extend a class from `JFrame` to create a GUI application.**

By now you know almost everything necessary to create an applet. That is because applets are very similar to GUI applications. You can think of an applet as a GUI application that runs under the control of a Web browser. Instead of displaying its own window, an applet appears in the browser's window. The differences between GUI application code and applet code are summarized here:

- A GUI application class inherits from `JFrame`. An applet class inherits from `JApplet`. The `JApplet` class is part of the `javax.swing` package.
- A GUI application class has a constructor that creates other components and sets up the GUI. An applet class does not normally have a constructor. Instead, it has a method named `init` that performs the same operations as a constructor. The `init` method accepts no arguments and has a `void` return type.
- The following methods, which are commonly called in a GUI application's constructor, are not called in an applet:

  `setTitle`

  `setSize`

  `setDefaultCloseOperation`

  `pack`

  `setVisible`

VideoNote
Creating an Applet

The methods listed here are used in a GUI application to affect the application's window in some way. They are not usually applicable to an applet because the applet does not have a window of its own.

- There is no static `main` method needed to create an instance of the applet class. The browser creates an instance of the class automatically.

Let's look at a simple applet. Code Listing 25-6 shows an applet that displays a label.

**Code Listing 25-6** `(SimpleApplet.java)`

```
import javax.swing.*;
import java.awt.*;

/**
   This is a simple applet.
*/

public class SimpleApplet extends JApplet
{
   /**
      The init method sets up the applet, much
      like a constructor.
   */

   public void init()
   {
      // Create a label.
      JLabel label =
           new JLabel("This is my very first applet.");

      // Set the layout manager.
      setLayout(new FlowLayout());

      // Add the label to the content pane.
      add(label);
   }
}
```

This code is very much like a regular GUI application. Although this class extends `JApplet` instead of `JFrame`, you still add components to the content pane and use layout managers in the same way.

## Running an Applet

The process of running an applet is different from that of running an application. To run an applet, you create an HTML document with an `applet` tag, which has the following general format:

```
<applet code="Filename.class" width=Wide height=High></applet>
```

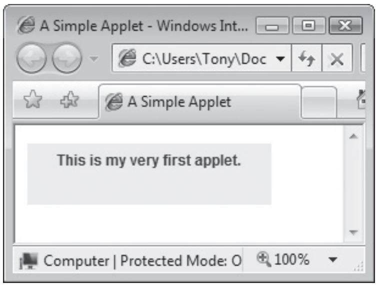
In the general format, ***`Filename.class`*** is the name of the applet's *.class* file. This is the file that contains the compiled byte code. Note that you do not specify the *.java* file, which contains the Java source code. You can optionally specify a path along with the file name. If you specify only the file name, it is assumed that the file is in the same directory as the HTML document. ***`Wide`*** is the width of the applet in pixels, and ***`High`*** is the height of the applet in pixels. When a browser processes an `applet` tag, it loads specified byte code and executes it in an area that is the size specified by the ***`Wide`*** and ***`High`*** values.

The HTML document shown in Code Listing 25-7 uses an `applet` tag to load the applet shown in Code Listing 25-6. This document specifies that the applet should be displayed in an area that is 200 pixels wide by 50 pixels high. Figure 25-12 shows this document when it is displayed in a Web browser.

**Code Listing 25-7** **(`SimpleApplet.html`)**

```
<html>
<head>
    <title>A Simple Applet</title>
</head>
<body>
    <applet code="SimpleApplet.class" width="200" height="50">
    </applet>
</body>
</html>
```

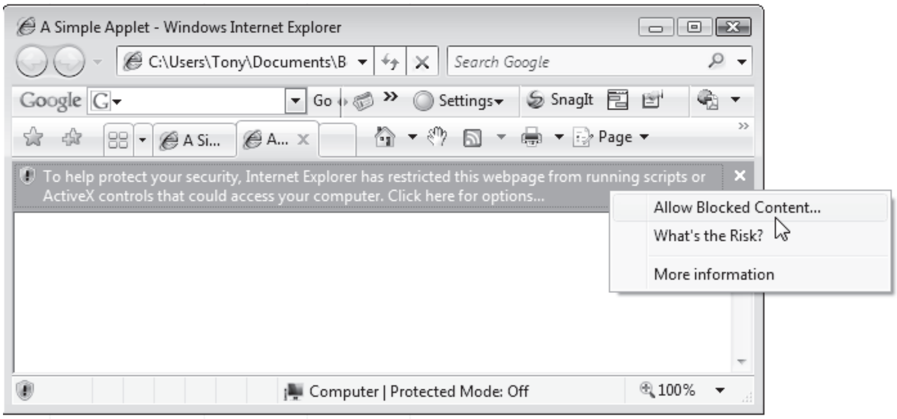
**Figure 25-12** The Web page produced by *SimpleApplet.html* (Microsoft Corporation)


A Simple Applet - Windows Int...
C:\Users\Tony\Doc
A Simple Applet
Computer | Protected Mode: O
100%

This is my very first applet.

**NOTE:** When you load a Web page that uses an applet into your browser, you will most likely get a security warning. For example, Figure 25-13 shows the warning you get from Internet Explorer. To run the applet, click the warning message and then select Allow Blocked Content . . . from the pop-up menu that appears.

**Figure 25-13** Security warning in Internet Explorer (Microsoft Corporation)


A Simple Applet - Windows Internet Explorer
C:\Users\Tony\Documents\B
Search Google
Google
Go
Settings
SnagIt
A Si...
A...
Page
Allow Blocked Content...
What's the Risk?
More information
Computer | Protected Mode: Off
100%

To help protect your security, Internet Explorer has restricted this webpage from running scripts or ActiveX controls that could access your computer. Click here for options...

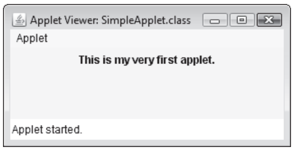
### Running an Applet with `appletviewer`

The Sun JDK comes with an applet viewer program that loads and executes an applet without the need for a Web browser. This program can be run from a command prompt with the `appletviewer` command. When you run the program, you specify the name of an HTML document as a command line argument. For example, the following command passes `SimpleApplet.html` as the command line argument:

```
appletviewer SimpleApplet.html
```

This command executes any applet that is referenced by an `applet` tag in the file *SimpleApplet.html*. The window shown in Figure 25-14 will be displayed.

**Figure 25-14** Applet executed by `appletviewer` (Oracle Corporate Counsel)



This is my very first applet.

Applet started.

> **NOTE:** The applet viewer does not display any output generated by text or tags in the HTML document. It only executes applets. If the applet viewer opens an HTML document with more than one `applet` tag, it will execute each applet in a separate window.

## Handling Events in an Applet

In an applet, events are handled with event listeners exactly as they are in GUI applications. To demonstrate, we will examine the `TempConverter` class, which is shown in Code Listing 25-8. This class is the applet displayed in the Web page we examined at the beginning of this chapter. It has a text field where the user can enter a Fahrenheit temperature and a Convert button that converts the temperature to Celsius and displays it in a read-only text field. The temperature conversion is performed in an action listener class that handles the button's action events.

**Code Listing 25-8** (`TempConverter.java`)

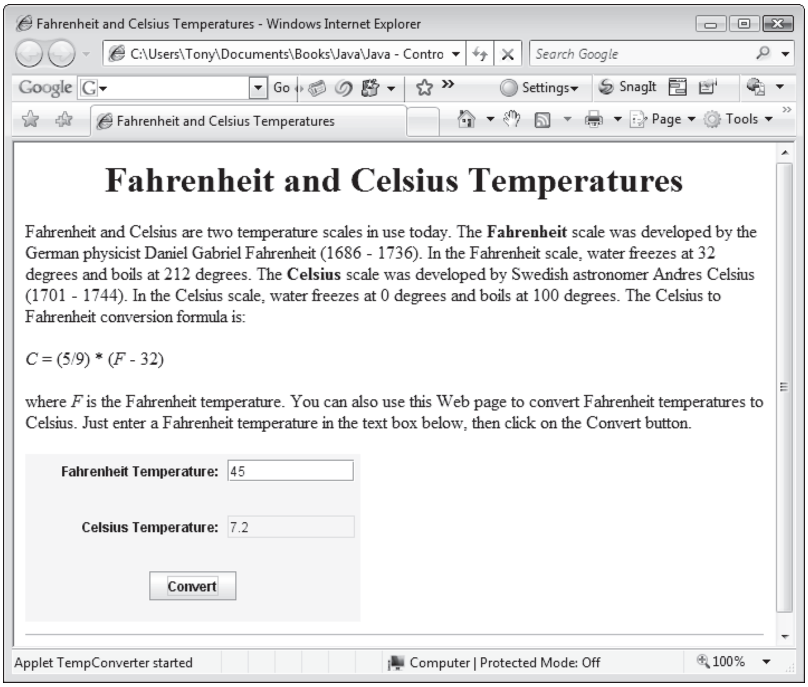
```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   The TempConverter class is an applet that converts
   Fahrenheit temperatures to Celsius.
*/

public class TempConverter extends JApplet
{
   private JPanel fPanel;         // To hold a text field
   private JPanel cPanel;         // To hold a text field
   private JPanel buttonPanel;    // To hold a button
   private JTextField fahrenheit; // Fahrenheit temperature
   private JTextField celsius;    // Celsius temperature

   /**
      init method
   */

   public void init()
   {
      // Build the panels.
      buildFpanel();
      buildCpanel();
      buildButtonPanel();

      // Create a layout manager.
      setLayout(new GridLayout(3, 1));

      // Add the panels to the content pane.
      add(fPanel);
      add(cPanel);
      add(buttonPanel);
   }

   /**
      The buildFpanel method creates a panel with a text
      field in which the user can enter a Fahrenheit
      temperature.
   */

   private void buildFpanel()
   {
      // Create the panel.
      fPanel = new JPanel();

      // Create a label to display a message.
      JLabel message1 =
                 new JLabel("Fahrenheit Temperature:");

      // Create a text field for the Fahrenheit temp.
      fahrenheit = new JTextField(10);

      // Create a layout manager for the panel.
      fPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));

      // Add the label and text field to the panel.
      fPanel.add(message1);
      fPanel.add(fahrenheit);
   }

   /**
      The buildCpanel method creates a panel that
      displays the Celsius temperature in a
      read-only text field.
   */

   private void buildCpanel()
   {
      // Create the panel.
      cPanel = new JPanel();

      // Create a label to display a message.
      JLabel message2 =
              new JLabel("Celsius Temperature:");

      // Create a text field for the celsius temp.
      celsius = new JTextField(10);

      // Make the text field read-only.
      celsius.setEditable(false);

      // Create a layout manager for the panel.
      cPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));

      // Add the label and text field to the panel.
      cPanel.add(message2);
      cPanel.add(celsius);
   }

   /**
      The buildButtonPanel method creates a panel with
      a button that converts the Fahrenheit temperature
      to Celsius.
   */

   private void buildButtonPanel()
   {
      // Create the panel.
      buttonPanel = new JPanel();

      // Create a button with the text "Convert".
      JButton convButton = new JButton("Convert");

      // Add an action listener to the button.
      convButton.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(convButton);
   }

   /**
      Private inner class that handles the action event
      that is generated when the user clicks the convert
      button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         double ftemp, ctemp;  // To hold the temperatures

         // Get the Fahrenheit temperature and convert it
         // to a double.
         ftemp = Double.parseDouble(fahrenheit.getText());

         // Calculate the Celsius temperature.
         ctemp = (5.0 / 9.0) * (ftemp - 32);

         // Display the Celsius temperature.
         celsius.setText(String.format("%.1f",  ctemp));
      }
   }
}
```

Code Listing 25-9 shows the contents of `TempConverter.html`, an HTML document that uses this applet. Figure 25-15 shows the Web page produced by this document. In the figure, the user has entered a Fahrenheit temperature and converted it to Celsius.

**Code Listing 25-9** (`TempConverter.html`)

```
<html>
<head>
    <title>Fahrenheit and Celsius Temperatures</title>
</head>
<body>
    <center>
        <h1>Fahrenheit and Celsius Temperatures</h1>
    </center>
    Fahrenheit and Celsius are two temperature scales in use today.
    The <b>Fahrenheit</b> scale was developed by the German physicist
    Daniel Gabriel Fahrenheit (1686 - 1736). In the Fahrenheit scale,
    water freezes at 32 degrees and boils at 212 degrees. The
    <b>Celsius</b> scale was developed by Swedish astronomer Andres Celsius
    (1701 - 1744). In the Celsius scale, water freezes at 0 degrees and
    boils at 100 degrees. The Celsius to Fahrenheit conversion formula
    is:
    <p />
    <i>C</i> = (5/9) * (<i>F</i> - 32)
    <p />
    where <i>F</i> is the Fahrenheit temperature. You can also use
    this Web page to convert Fahrenheit temperatures to Celsius.
    Just enter a Fahrenheit temperature in the text box below, then
    click on the Convert button.
    <p />
    <applet code="TempConverter.class" width="300" height="150">
    </applet>
    <hr />
</body>
</html>
```

**Figure 25-15** Web page produced by *TempConverter.html* (Microsoft Corporation)


Fahrenheit and Celsius Temperatures - Windows Internet Explorer
C:\Users\Tony\Documents\Books\Java\Java - Contro
Search Google
Google
Go
Settings
SnagIt
Fahrenheit and Celsius Temperatures
Page
Tools
Applet TempConverter started
Computer | Protected Mode: Off
100%

Fahrenheit and Celsius Temperatures

Fahrenheit and Celsius are two temperature scales in use today. The **Fahrenheit** scale was developed by the German physicist Daniel Gabriel Fahrenheit (1686 - 1736). In the Fahrenheit scale, water freezes at 32 degrees and boils at 212 degrees. The **Celsius** scale was developed by Swedish astronomer Andres Celsius (1701 - 1744). In the Celsius scale, water freezes at 0 degrees and boils at 100 degrees. The Celsius to Fahrenheit conversion formula is:

$C = (5/9) * (F - 32)$

where $F$ is the Fahrenheit temperature. You can also use this Web page to convert Fahrenheit temperatures to Celsius. Just enter a Fahrenheit temperature in the text box below, then click on the Convert button.

Fahrenheit Temperature: 45

Celsius Temperature: 7.2

Convert

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

25.12 Instead of `JFrame`, an applet class is extended from what class?

25.13 Instead of a constructor, an applet class uses what method?

25.14 Why is there no need for a static `main` method to create an instance of an applet class?

25.15 Suppose the file *MyApplet.java* contains the Java source code for an applet. What tag would you write in an HTML document to run the applet in an area that is 400 pixels wide by 200 pixels high?

## 25.4 Using AWT for Portability

**CONCEPT: Applets that use Swing components may be incompatible with some browsers. If you want to make sure that an applet is compatible with all Java-enabled browsers, use AWT components instead of Swing.**

Java provides two libraries of classes that GUI components may be created from. Recall from Chapter 17 that these libraries are AWT and Swing. AWT is the original library that has been part of Java since its earliest version. Swing is an improved library that was introduced with Java 2. All of the GUI applications in Chapters 17 and 18, as well as the applets we have studied so far in this chapter, use Swing classes for their components.

Some browsers, do not directly support the Swing classes in applets. These browsers require a *plug-in*, which is software that extends or enhances another program, in order to run applets that use Swing components. Fortunately, this plug-in is automatically installed on a computer when the Sun JDK is installed. If you have installed the JDK, you should be able to write applets that use Swing and run them with no problems.

If you are writing an applet for other people to run on their computers, however, there is no guarantee that they will have the required plug-in. If this is the case, you should use the AWT classes instead of the Swing classes for the components in your applet. Fortunately, the AWT component classes are very similar to the Swing classes, so learning to use them is simple if you already know how to use Swing.

There is a corresponding AWT class for each of the Swing classes that you have learned so far. The names of the AWT classes are the same as those of the Swing classes, except the AWT class names do not start with the letter `J`. For example, the AWT class to create a frame is named `Frame`, and the AWT class to create a panel is named `Panel`. Table 25-1 lists several of the AWT classes. All of these classes are in the `java.awt` package.

**Table 25-1** Several AWT classes

| AWT Class | Description | Corresponding Swing Class |
|---|---|---|
| `Applet` | Used as a superclass for all applets. Unlike `JApplet` objects, `Applet` objects do not have a content pane. | `JApplet` |
| `Frame` | Creates a frame container that may be displayed as a window. Unlike `JFrame` objects, `Frame` objects do not have a content pane. | `JFrame` |
| `Panel` | Creates a panel container. | `JPanel` |
| `Button` | Creates a button that may be clicked. | `JButton` |
| `Label` | Creates a label that displays text. | `JLabel` |
| `TextField` | Creates a single line text field, which the user may type into. | `JTextField` |
| `Checkbox` | Creates a check box that may be selected or deselected. | `JCheckBox` |

The Swing classes were intentionally designed with constructors and methods that are similar to those of their AWT counterparts. In addition, events are handled in the same way for each set of classes. This makes it easy for you to use either set of classes without learning a completely different syntax for each. For example, Code Listing 25-10 shows a version of the `TempConverter` applet that has been rewritten to use AWT components instead of Swing components.

**Code Listing 25-10** **(`AWTTempConverter.java`)**

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/**
   The AWTTempConverter class is an applet that converts
   Fahrenheit temperatures to Celsius.
*/

public class AWTTempConverter extends Applet
{
   private Panel fPanel;          // To hold a text field
   private Panel cPanel;          // To hold a text field
   private Panel buttonPanel;     // To hold a button
   private TextField fahrenheit;  // Fahrenheit temperature
   private TextField celsius;     // Celsius temperature

   /**
      init method
   */

   public void init()
   {
      // Build the panels.
      buildFpanel();
      buildCpanel();
      buildButtonPanel();

      // Create a layout manager.
      setLayout(new GridLayout(3, 1));

      // Add the panels to the applet.
      add(fPanel);
      add(cPanel);
      add(buttonPanel);
   }

   /**
      The buildFpanel method creates a panel with a text
      field in which the user can enter a Fahrenheit
      temperature.
   */

   private void buildFpanel()
   {
      // Create the panel.
      fPanel = new Panel();

      // Create a label to display a message.
      Label message1 = new Label("Fahrenheit Temperature:");

      // Create a text field for the Fahrenheit temp.
      fahrenheit = new TextField(10);

      // Create a layout manager for the panel.
      fPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));

      // Add the label and text field to the panel.
      fPanel.add(message1);
      fPanel.add(fahrenheit);
   }

   /**
      The buildCpanel method creates a panel that
      displays the Celsius temperature in a
      read-only text field.
   */

   private void buildCpanel()
   {
      // Create the panel.
      cPanel = new Panel();

      // Create a label to display a message.
      Label message2 = new Label("Celsius Temperature:");

      // Create a text field for the Celsius temp.
      celsius = new TextField(10);

      // Make the text field read-only.
      celsius.setEditable(false);

      // Create a layout manager for the panel.
      cPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));

      // Add the label and text field to the panel.
      cPanel.add(message2);
      cPanel.add(celsius);
   }

   /**
      The buildButtonPanel method creates a panel with
      a button that converts the Fahrenheit temperature
      to Celsius.
   */


   private void buildButtonPanel()
   {
      // Create the panel.
      buttonPanel = new Panel();

      // Create a button with the text "Convert".
      Button convButton = new Button("Convert");

      // Add an action listener to the button.
      convButton.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(convButton);
   }

   /**
      Private inner class that handles the action event
      that is generated when the user clicks the convert
      button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         double ftemp, ctemp;  // To hold the temperatures

         // Get the Fahrenheit temperature and convert it
         // to a double.
         ftemp = Double.parseDouble(fahrenheit.getText());

         // Calculate the Celsius temperature.
         ctemp = (5.0 / 9.0) * (ftemp - 32);

         // Display the Celsius temperature.
         celsius.setText(String.format("%.1f", ctemp));
      }
   }
}
```

The only modifications that were made were as follows:

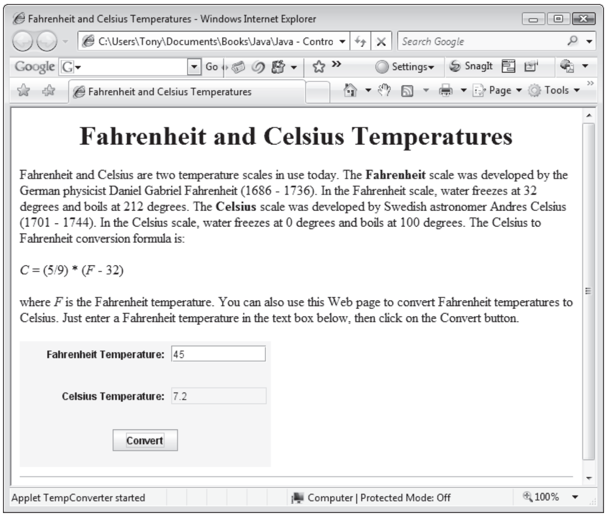
- The `JApplet`, `JPanel`, `JLabel`, `JTextField`, and `JButton` classes were replaced with the `Applet`, `Panel`, `Label`, `TextField`, and `Button` classes.
- The `import javax.swing.*;` statement was removed.

To run the applet in a browser, the `APPLET` tag in the *TempConverter.html* file must be modified to read as follows:

```
<applet code="AWTTempConverter.class" width=300 height=150>
</applet>
```

Once this change is made, the *TempConverter.html* file produces the Web page shown in Figure 25-16.

**Figure 25-16** Web page running the `AWTTempConverter` applet (Microsoft Corporation)


Fahrenheit and Celsius Temperatures - Windows Internet Explorer
C:\Users\Tony\Documents\Books\Java\Java - Contro
Search Google
Google
Go
Settings
SnagIt
Fahrenheit and Celsius Temperatures
Page
Tools

Fahrenheit and Celsius Temperatures

Fahrenheit and Celsius are two temperature scales in use today. The **Fahrenheit** scale was developed by the German physicist Daniel Gabriel Fahrenheit (1686 - 1736). In the Fahrenheit scale, water freezes at 32 degrees and boils at 212 degrees. The **Celsius** scale was developed by Swedish astronomer Andres Celsius (1701 - 1744). In the Celsius scale, water freezes at 0 degrees and boils at 100 degrees. The Celsius to Fahrenheit conversion formula is:

$C = (5/9) * (F - 32)$

where $F$ is the Fahrenheit temperature. You can also use this Web page to convert Fahrenheit temperatures to Celsius. Just enter a Fahrenheit temperature in the text box below, then click on the Convert button.

Fahrenheit Temperature: 45

Celsius Temperature: 7.2

Convert
Applet TempConverter started
Computer | Protected Mode: Off
100%

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

25.16 To create an applet using AWT, what class do you inherit your applet class from?

25.17 In Swing, if an object's class extends `JFrame` or `JApplet`, you add components to its content pane. How do you add components to an object if its class extends `Frame` or `Applet`?

## 25.5 Drawing Shapes

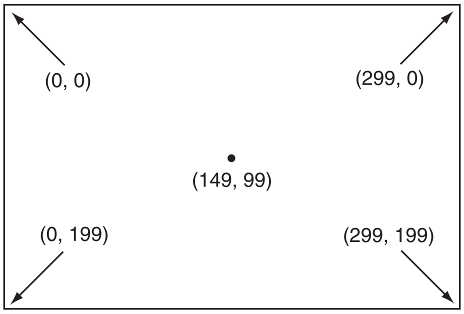
**CONCEPT: Components have an associated `Graphics` object that may be used to draw lines and shapes.**

In addition to displaying standard components such as buttons and labels, Java allows you to draw lines and graphical shapes such as rectangles, ovals, and arcs. These lines and shapes are drawn directly on components. This allows a frame or a panel to become a canvas for your drawings. Before we examine how to draw graphics on a component, however, we must discuss the *XY* coordinate system. You use the *XY* coordinate system to specify the location of your graphics.

### The *XY* Coordinate System

The location of each pixel in a component is identified with an *X* coordinate and a *Y* coordinate. The coordinates are usually written in the form (*X*, *Y*). The *X* coordinate identifies a pixel's horizontal location, and the *Y* coordinate identifies its vertical location. The coordinates of the pixel in the upper-left corner of a component are usually (0, 0). The *X* coordinates increase from left to right, and the *Y* coordinates increase from top to bottom. For example, Figure 25-17 illustrates a component such as a frame or a panel that is 300 pixels wide by 200 pixels high. The *X* and *Y* coordinates of the pixels in each corner, as well as the pixel in the center of the component are shown. The pixel in the center of the component has an *X* coordinate of 149 and a *Y* component of 99.

**Figure 25-17** *X* and *Y* coordinates on a 300 pixel wide by 200 pixel high component



When you draw a line or shape on a component, you must indicate its position using *X* and *Y* coordinates.

### Graphics Objects

Each component has an internal object that inherits from the `Graphics` class, which is part of the `java.awt` package. This object has numerous methods for drawing graphical shapes on the surface of the component. Table 25-2 lists some of these methods.

**Table 25-2** Some of the `Graphics` class methods

| Method | Description |
|---|---|
| `void setColor(Color c)` | Sets the drawing color for this object to that specified by the argument. |
| `Color getColor()` | Returns the current drawing color for this object. |
| `void drawLine(int x1, int y1, int x2, int y2)` | Draws a line on the component starting at the coordinate (`x1`, `y1`) and ending at the coordinate (`x2`, `y2`). The line will be drawn in the current drawing color. |
| `void drawRect(int x, int y, int width, int height)` | Draws the outline of a rectangle on the component. The upper-left corner of the rectangle will be at the coordinate (`x`, `y`). The `width` parameter specifies the rectangle's width in pixels, and `height` specifies the rectangle's height in pixels. The rectangle will be drawn in the current drawing color. |
| `void fillRect(int x, int y, int width, int height)` | Draws a filled rectangle. The parameters are the same as those used by the `drawRect` method. The rectangle will be filled with the current drawing color. |
| `void drawOval(int x, int y, int width, int height)` | Draws the outline of an oval on the component. The shape and size of the oval is determined by an invisible rectangle that encloses it. The upper-left corner of the rectangle will be at the coordinate (`x`, `y`). The `width` parameter specifies the rectangle's width in pixels, and `height` specifies the rectangle's height in pixels. The oval will be drawn in the current drawing color. |
| `void fillOval(int x, int y, int width, int height)` | Draws a filled oval. The parameters are the same as those used by the `drawOval` method. The oval will be filled in the current drawing color. |
| `void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` | This method draws an arc, which is considered to be part of an oval. The shape and size of the oval are determined by an invisible rectangle that encloses it. The upper-left corner of the rectangle will be at the coordinate (`x`, `y`). The `width` parameter specifies the rectangle's width in pixels, and `height` specifies the rectangle's height in pixels. The arc begins at the angle `startAngle`, and ends at the angle `arcAngle`. The arc will be drawn in the current drawing color. |
| `void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` | This method draws a filled arc. The parameters are the same as those used by the `drawArc` method. The arc will be filled with the current drawing color. |

*(table continues next page)*

**Table 25-2** Some of the `Graphics` class methods (continued)

| Method | Description |
|---|---|
| `void drawPolygon(int[] xPoints,`<br>`int[] yPoints,`<br>`int numPoints)` | This method draws the outline of a closed polygon on the component. The `xPoints` array contains the *X*-coordinates for each vertex, and the `yPoints` array contains the *Y* coordinates for each vertex. The argument passed into `numPoints` is the number of vertices in the polygon. |
| `void fillPolygon(int[] xPoints,`<br>`int[] yPoints,`<br>`int numPoints)` | This method draws a filled polygon. The parameters are the same as those used by the `drawPolygon` method. The polygon will be filled with the current drawing color. |
| `void drawstring(String str,`<br>`int x, int y)` | Draws the string passed into `str` using the current font. The bottom left of the string is drawn at the coordinates passed into `x` and `y`. |
| `void setFont(Font f)` | Sets the current font, which is used by the `drawString` method. |

In order to call any of these methods, you must get a reference to a component's `Graphics` object. One way to do this is to override the `paint` method. You can override the `paint` method in any class that extends as follows:

- `JApplet`
- `JFrame`
- Any AWT class, including `Applet` and `Frame`

The `paint` method is responsible for displaying, or "painting," a component on the screen. This method is automatically called when the component is first displayed and is called again any time the component needs to be redisplayed. For example, when the component is completely or partially obscured by another window, and the obscuring window is moved, then the component's `paint` method is called to redisplay it. The header for the `paint` method is:

```
public void paint(Graphics g)
```

Notice that the method's argument is a `Graphics` object. When this method is called for a particular component, the `Graphics` object that belongs to that component is automatically passed as an argument. By overriding the `paint` method, you can use the `Graphics` object argument to draw your own graphics on the component. For example, look at the applet class in Code Listing 25-11.

This class inherits from `JApplet`, and it overrides the `paint` method. The `Graphics` object that is passed into the `paint` method's `g` parameter is the object that is responsible for drawing the entire applet window. Notice that in line 29 the method first calls the superclass version of the `paint` method, passing the object `g` as an argument. When overriding the `paint` method, you should always call the superclass's `paint` method before doing anything else. This ensures that the component will be displayed properly on the screen.

**Code Listing 25-11** (`LineDemo.java`)

```java
import javax.swing.*;
import java.awt.*;

/**
   This class is an applet that demonstrates how lines
   can be drawn.
*/

public class LineDemo extends JApplet
{
   /**
      init method
   */

   public void init()
   {
      // Set the background color to white.
      getContentPane().setBackground(Color.white);
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass paint method.
      super.paint(g);

      // Draw a red line from (20, 20) to (280, 280).
      g.setColor(Color.red);
      g.drawLine(20, 20, 280, 280);

      // Draw a blue line from (280, 20) to (20, 280).
      g.setColor(Color.blue);
      g.drawLine(280, 20, 20, 280);
   }
}
```

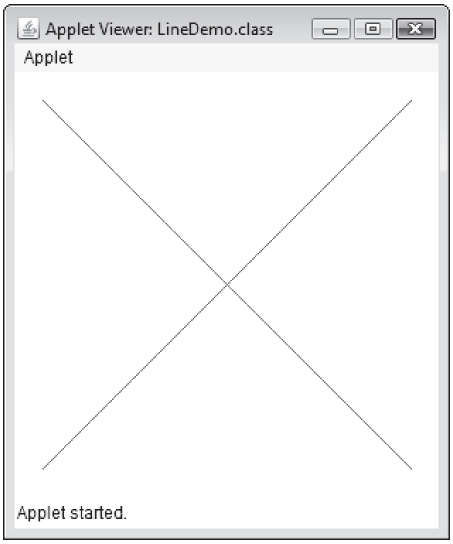
In line 32 the method sets the drawing color to red. In line 33 a line is drawn from the coordinates (20, 20) to (280, 280). This is a diagonal line drawn from the top-left area of the applet window to the bottom-right area. Next, in line 36, the drawing color is set to blue. In line 37 a line is drawn from (280, 20) to (20, 280). This is also a diagonal line. It is drawn from the top-right area of the applet window to the bottom-left area.

We can use the *LineDemo.html* file, which is in the same folder as the applet class, to execute the applet. The following line in the file runs the applet in an area that is 300 pixels wide by 300 pixels high:

```
<applet code="LineDemo.class" width=300 height=300>
</applet>
```

Figure 25-18 shows the applet running in the applet viewer.

**Figure 25-18** `LineDemo` applet (Oracle Corporate Counsel)



Notice that the `paint` method is not explicitly called by the applet. It is automatically called when the applet first executes. As previously mentioned, it is also called any time the applet window needs to be redisplayed.

Code Listing 25-12 shows the `RectangleDemo` class, an applet that draws two rectangles: one as a black outline and one filled with red. Each rectangle is 120 pixels wide and 120 pixels high. The file *RectangleDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

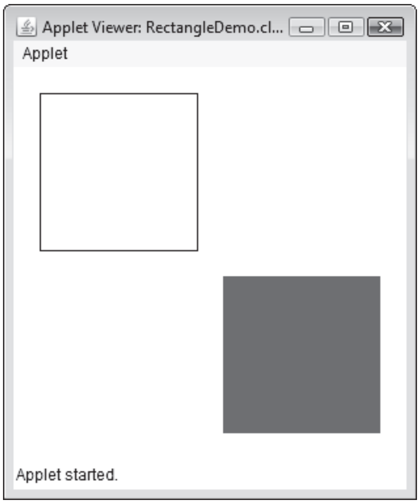
```
<applet code="RectangleDemo.class" width=300 height=300>
</applet>
```

Figure 25-19 shows the applet running in the applet viewer.

**Code Listing 25-12** (`RectangleDemo.java`)

```
import javax.swing.*;
import java.awt.*;

/**
   This class is an applet that demonstrates how
   rectangles can be drawn.
*/

public class RectangleDemo extends JApplet
{
   /**
      init method
   */

   public void init()
   {
      // Set the background color to white.
      getContentPane().setBackground(Color.white);
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass paint method.
      super.paint(g);

      // Draw a black unfilled rectangle.
      g.setColor(Color.black);
      g.drawRect(20, 20, 120, 120);

      // Draw a red filled rectangle.
      g.setColor(Color.red);
      g.fillRect(160, 160, 120, 120);
   }
}
```

**Figure 25-19** `RectangleDemo` applet (Oracle Corporate Counsel)



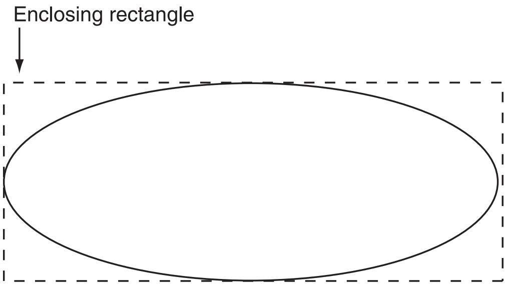
Code Listing 25-13 shows the `OvalDemo` class, an applet that draws two ovals. An oval is enclosed in an invisible rectangle that establishes the boundaries of the oval. The width and height of the enclosing rectangle defines the shape and size of the oval. This is illustrated in Figure 25-20.

When you call the `drawOval` or `fillOval` method, you pass the *X* and *Y* coordinates of the enclosing rectangle's upper-left corner, and the width and height of the enclosing rectangle as arguments.

**Code Listing 25-13** (`OvalDemo.java`)

```
import javax.swing.*;
import java.awt.*;

/**
   This class is an applet that demonstrates how
   ovals can be drawn.
*/

public class OvalDemo extends JApplet
{
   /**
      init method
   */

   public void init()
   {
      // Set the background color to white.
      getContentPane().setBackground(Color.white);
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass paint method.
      super.paint(g);

      // Draw a black unfilled oval.
      g.setColor(Color.black);
      g.drawOval(20, 20, 120, 75);

      // Draw a green filled oval.
      g.setColor(Color.green);
      g.fillOval(80, 160, 180, 75);
   }
}
```

**Figure 25-20** An oval and its enclosing rectangle

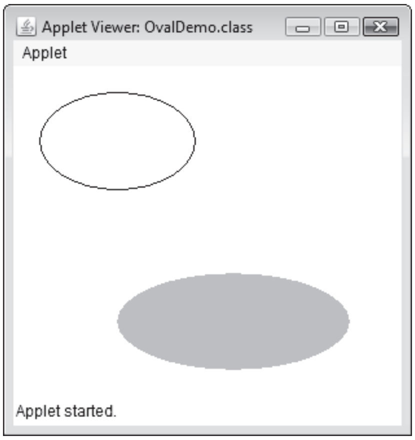


The file *OvalDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="OvalDemo.class" width=300 height=255>
</applet>
```

Figure 25-21 shows the applet running in the applet viewer.

**Figure 25-21** `OvalDemo` applet (Oracle Corporate Counsel)



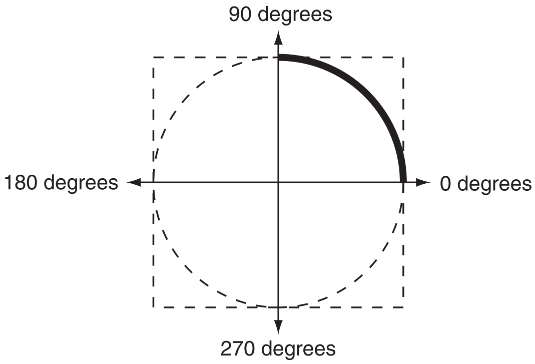
**TIP:** To draw a circle, simply draw an oval with an enclosing rectangle that is square. In other words, the enclosing rectangle's width and height should be the same.

The `drawArc` method draws an arc, which is part of an oval. You pass the same arguments to `drawArc` as you do to `drawOval`, plus two additional arguments: the arc's starting angle and ending angle. The angles are measured in degrees, with 0 degrees being at the 3 o'clock position. For example, look at the following statement:

```
g.drawArc(20, 20, 100, 100, 0, 90);
```

This statement creates an enclosing rectangle with its upper-left corner at (20, 20) and with a width and height of 100 pixels each. The oval constructed from this enclosing rectangle is a circle. The arc that is drawn is the part of the oval that starts at 0 degrees and ends at 90 degrees. Figure 25-22 illustrates this arc. The dashed lines show the enclosing rectangle and the oval. The thick black line shows the arc that will be drawn.

**Figure 25-22** An arc



Code Listing 25-14 shows the `ArcDemo` class, which is an applet that draws four arcs: two unfilled and two filled. The filled arcs are drawn with the `fillArc` method.

The file *ArcDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="ArcDemo.class" width=300 height=220>
</applet>
```

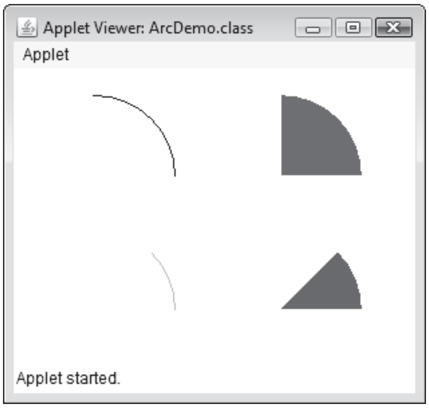
Figure 25-23 shows the applet running in the applet viewer.

**Code Listing 25-14** (`ArcDemo.java`)

```
import javax.swing.*;
import java.awt.*;

/**
   This class is an applet that demonstrates how
   arcs can be drawn.
*/

public class ArcDemo extends JApplet
{
   /**
      init method
   */

   public void init()
   {
      // Set the background color to white.
      getContentPane().setBackground(Color.white);
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass paint method.
      super.paint(g);

      // Draw a black unfilled arc from 0 degrees
      // to 90 degrees.
      g.setColor(Color.black);
      g.drawArc(0, 20, 120, 120, 0, 90);

      // Draw a red filled arc from 0 degrees
      // to 90 degrees.
```

```
        g.setColor(Color.red);
        g.fillArc(140, 20, 120, 120, 0, 90);

        // Draw a green unfilled arc from 0 degrees
        // to 45 degrees.
        g.setColor(Color.green);
        g.drawArc(0, 120, 120, 120, 0, 45);

        // Draw a blue filled arc from 0 degrees
        // to 45 degrees.
        g.setColor(Color.blue);
        g.fillArc(140, 120, 120, 120, 0, 45);
    }
}
```

**Figure 25-23** ArcDemo applet (Oracle Corporate Counsel)



Applet started.

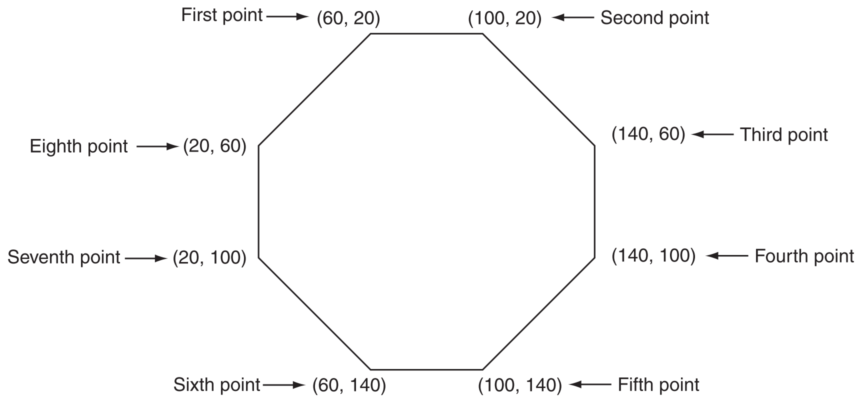
The `drawPolygon` method draws an outline of a closed polygon and the `fillPolygon` method draws a closed polygon filled with the current drawing color. A polygon is constructed of multiple line segments that are connected. The point where two line segments are connected is called a *vertex*. These methods accept two `int` arrays as arguments. The first array contains the *X* coordinates of each vertex, and the second array contains the *Y* coordinates of each vertex. The third argument is an `int` that specifies the number of vertices, or connecting points.

For example, suppose we use the following arrays as arguments for the *X* and *Y* coordinates of a polygon:

```
int[] xCoords = {60, 100, 140, 140, 100, 60, 20, 20 };
int[] yCoords = {20, 20, 60, 100, 140, 140, 100, 60 };
```

The first point specified by these arrays is (60, 20), the second point is (100, 20), and so forth. There are a total of eight points specified by these arrays, and if we connect each of these points we get the octagon shown in Figure 25-24.

**Figure 25-24** Points of each vertex in an octagon


First point
(60, 20)
(100, 20)
Second point
Eighth point
(20, 60)
(140, 60)
Third point
Seventh point
(20, 100)
(140, 100)
Fourth point
Sixth point
(60, 140)
(100, 140)
Fifth point

If the last point specified in the arrays is different from the first point, as in this example, then the two points are automatically connected to close the polygon. The `PolygonDemo` class in Code Listing 25-15 draws a filled polygon using these arrays as arguments.

**Code Listing 25-15** (`PolygonDemo.java`)

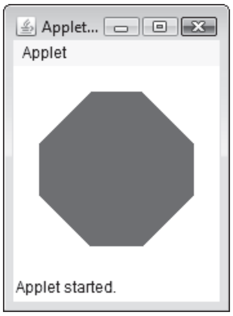
```
import javax.swing.*;
import java.awt.*;

/**
   This class is an applet that demonstrates how a
   polygon can be drawn.
*/

public class PolygonDemo extends JApplet
{
   /**
      init method
   */

   public void init()
   {
      // Set the background color to white.
      getContentPane().setBackground(Color.white);
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      int[] xCoords = {60, 100, 140, 140,
                       100, 60, 20, 20 };
      int[] yCoords = {20, 20, 60, 100,
                       140, 140, 100, 60 };

      // Call the superclass paint method.
      super.paint(g);

      // Set the drawing color.
      g.setColor(Color.red);

      // Draw the polygon.
      g.fillPolygon(xCoords, yCoords, 8);
   }
}
```

The file *PolygonDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="PolygonDemo.class" width=160 height=160>
</applet>
```

Figure 25-25 shows the applet running in the applet viewer.

**Figure 25-25** PolygonDemo applet (Oracle Corporate Counsel)



Applet started.

The `drawString` method draws a string as a graphic. The string is specified by its first argument, a `String` object. The *X* and *Y* coordinates of the lower-left point of the string are specified by the second and third arguments. For example, assuming that `g` references a `Graphics` object, the following statement draws the string `"Hello World"`, starting at the coordinates 100, 50:

```
g.drawstring("Hello World", 100, 50);
```

You can set the font for the string with the `setFont` method. This method accepts a `Font` object as its argument. Here is an example:

```
g.setFont(new Font("Serif", Font.ITALIC, 20));
```

The `Font` class was covered in Chapter 18. Recall that the `Font` constructor's arguments are the name of a font, the font's style, and the font's size in points. You can combine font styles with the + operator, as follows:

```
g.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

The `GraphicStringDemo` class in Code Listing 25-16 demonstrates the `drawString` method. It draws the same octagon that the `PolygonDemo` class drew, and then draws the string `"STOP"` over it to create a stop sign. The string is drawn in a bold 35-point san serif font.

**Code Listing 25-16** (`GraphicStringDemo.java`)

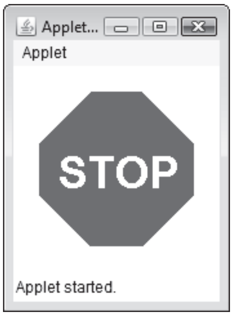
```
import javax.swing.*;
import java.awt.*;

/**
   This class is an applet that demonstrates how a
   string can be drawn.
*/

public class GraphicStringDemo extends JApplet
{
   /**
      init method
   */

   public void init()
   {
      // Set the background color to white.
      getContentPane().setBackground(Color.white);
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      int[] xCoords = {60, 100, 140, 140,
                       100, 60, 20, 20 };
      int[] yCoords = {20, 20, 60, 100,
                       140, 140, 100, 60 };

      // Call the superclass paint method.
      super.paint(g);

      // Set the drawing color.
      g.setColor(Color.red);

      // Draw the polygon.
      g.fillPolygon(xCoords, yCoords, 8);

      // Set the drawing color to white.
      g.setColor(Color.white);

      // Set the font and draw "STOP".
      g.setFont(new Font("SansSerif", Font.BOLD, 35));
      g.drawString("STOP", 35, 95);
   }
}
```

The file *GraphicStringDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="GraphicStringDemo.class" width=160 height=160>
</applet>
```

Figure 25-26 shows the applet running in the applet viewer.

**Figure 25-26** `GraphicStringDemo` applet (Oracle Corporate Counsel)



Applet started.

## The `repaint` Method

As previously mentioned, you do not call a component's `paint` method. It is automatically called when the component must be redisplayed. Sometimes, however, you might want to force the application or applet to call the `paint` method. You do this by calling the `repaint` method, which has the following header:

```
public void repaint()
```

The `repaint` method clears the surface of the component and then calls the `paint` method. You will see an applet that uses this method in a moment.

## Drawing on Panels

Each of the preceding examples uses the entire `JApplet` window as a canvas for drawing. Sometimes, however, you might want to confine your drawing space to a smaller region within the window, such as a panel. To draw on a panel, you simply get a reference to the panel's `Graphics` object and then use that object's methods to draw. The resulting graphics are drawn only on the panel.

Getting a reference to a `JPanel` component's `Graphics` object is similar to the technique you saw in the previous examples. Instead of overriding the `JPanel` object's `paint` method, however, you should override its `paintComponent` method. This is true not only for `JPanel` objects, but also for all Swing components except `JApplet` and `JFrame`. The `paintComponent` method serves for `JPanel` and most other Swing objects the same purpose as the `paint` method: It is automatically called when the component needs to be redisplayed. When it is called, the component's `Graphics` object is passed as an argument. Here is the method's header:

```
public void paintComponent(Graphics g)
```

When you override this method, first you should call the superclass's `paintComponent` method to ensure that the component is properly displayed. Here is an example call to the superclass's version of the method:

```
super.paintComponent(g);
```

After this you can call any of the `Graphics` object's methods to draw on the component. As an example, we look at the `GraphicsWindow` class in Code Listing 25-17. When this applet is run (via the *GraphicsWindow.html* file, which is in the same folder as the applet class), the window shown in Figure 25-27 is displayed. A set of check boxes is displayed in a `JPanel` component on the right side of the window. The white area that occupies the majority of the window is a `DrawingPanel` object. The `DrawingPanel` class inherits from `JPanel`, and its code is shown in Code Listing 25-18. When one of the check boxes is selected, a shape appears in the `DrawingPanel` object. Figure 25-28 shows how the applet window appears when all of the check boxes are selected.

**Figure 25-27** `GraphicsWindow` applet (Oracle Corporate Counsel)

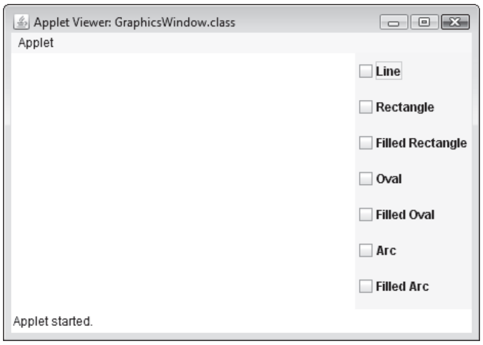
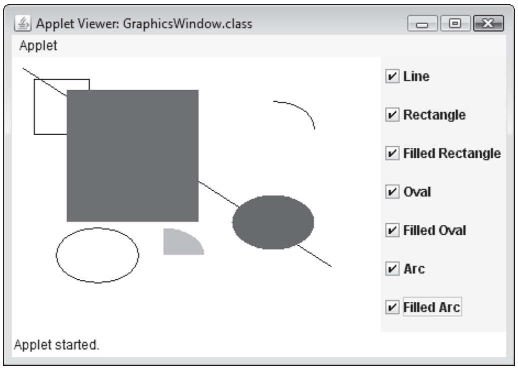


**Figure 25-28** GraphicsWindow applet with all graphics selected (Oracle Corporate Counsel)


Applet Viewer: GraphicsWindow.class
Applet
☑ Line
☑ Rectangle
☑ Filled Rectangle
☑ Oval
☑ Filled Oval
☑ Arc
☑ Filled Arc
Applet started.

**Code Listing 25-17** (GraphicsWindow.java)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   This class displays a drawing panel and a set of
   check boxes that allow the user to select shapes.
   The selected shapes are drawn on the drawing panel.
*/

public class GraphicsWindow extends JApplet
{
   // Declare an array of check box components
   private JCheckBox[] checkBoxes;

   // The following titles array contains the
   // titles of the check boxes.
   private String[] titles = { "Line", "Rectangle",
                               "Filled Rectangle",
                               "Oval", "Filled Oval",
                               "Arc", "Filled Arc" };

   // The following will reference a panel to contain
   // the check boxes.
   private JPanel checkBoxPanel;

   // The following will reference an instance of the
   // DrawingPanel class. This will be a panel to draw on.
   private DrawingPanel drawingPanel;

   /**
      init method
   */

   public void init()
   {
      // Build the check box panel.
      buildCheckBoxPanel();

      // Create the drawing panel.
      drawingPanel = new DrawingPanel(checkBoxes);

      // Add the check box panel to the east region
      // and the drawing panel to the center region.
      add(checkBoxPanel, BorderLayout.EAST);
      add(drawingPanel, BorderLayout.CENTER);
   }

   /**
      The buildCheckBoxPanel method creates the array of
      check box components and adds them to a panel.
   */

   private void buildCheckBoxPanel()
   {
      // Create the panel.
      checkBoxPanel = new JPanel();
      checkBoxPanel.setLayout(new GridLayout(7, 1));

      // Create the check box array.
      checkBoxes = new JCheckBox[7];

      // Create the check boxes and add them to the panel.
      for (int i = 0; i < checkBoxes.length; i++)
      {
         checkBoxes[i] = new JCheckBox(titles[i]);
         checkBoxes[i].addItemListener(
                                 new CheckBoxListener());
         checkBoxPanel.add(checkBoxes[i]);
      }
   }

   /**
      A private inner class to respond to changes in the
      state of the check boxes.
   */

   private class CheckBoxListener implements ItemListener
   {
      public void itemStateChanged(ItemEvent e)
      {
         drawingPanel.repaint();
      }
   }
}
```

**Code Listing 25-18** `(DrawingPanel.java)`

```
import javax.swing.*;
import java.awt.*;

/**
   This class creates a panel that example shapes are
   drawn on.
*/

public class DrawingPanel extends JPanel
{
   // Declare a check box array.
   private JCheckBox[] checkBoxArray;

   /**
      Constructor
   */

   public DrawingPanel(JCheckBox[] cbArray)
   {
      // Reference the check box array.
      checkBoxArray = cbArray;

      // Set the background color to white.
      setBackground(Color.white);

      // Set the preferred size of the panel.
      setPreferredSize(new Dimension(300, 200));
   }

   /**
      paintComponent method
      @param g The panel's Graphics object.
   */

   public void paintComponent(Graphics g)
   {
      // Call the superclass paintComponent method.
      super.paintComponent(g);

      // Draw the selected shapes.
      if (checkBoxArray[0].isSelected())
      {
         g.setColor(Color.black);
         g.drawLine(10, 10, 290, 190);
      }
      if (checkBoxArray[1].isSelected())
      {
         g.setColor(Color.black);
         g.drawRect(20, 20, 50, 50);
      }
      if (checkBoxArray[2].isSelected())
      {
         g.setColor(Color.red);
         g.fillRect(50, 30, 120, 120);
      }
      if (checkBoxArray[3].isSelected())
      {
         g.setColor(Color.black);
         g.drawOval(40, 155, 75, 50);
      }
      if (checkBoxArray[4].isSelected())
      {
         g.setColor(Color.blue);
         g.fillOval(200, 125, 75, 50);
      }
      if (checkBoxArray[5].isSelected())
      {
         g.setColor(Color.black);
         g.drawArc(200, 40, 75, 50, 0, 90);
      }
      if (checkBoxArray[6].isSelected())
      {
         g.setColor(Color.green);
         g.fillArc(100, 155, 75, 50, 0, 90);
      }
   }
}
```

Let's take a closer look at the applet's code. First, notice in lines 14 through 21 of the `GraphicsWindow` class (in Code Listing 25-17) that two of the class's fields are array reference variables. The `checkBoxes` variable references an array of `JCheckBox` components, and the `titles` variable references an array of strings. The strings in the `titles` array are the titles that the check boxes will display.

The first statement in the `init` method, line 38, is a call to the `buildCheckBoxPanel` method, which creates a panel for the check boxes, creates the array of check boxes, adds an item listener to each element of the array, and adds each element to the panel.

After the `buildCheckBoxPanel` method executes, the `init` method creates a `DrawingPanel` object with the statement in line 41. Notice that the `checkBoxes` variable is passed to the `DrawingPanel` constructor. The `drawingPanel` object needs a reference to the array so its `paintComponent` method can determine which check boxes are selected and draw the corresponding shape.

The only times that the `paintComponent` method is automatically called is when the component is initially displayed and when the component needs to be redisplayed. In order to display a shape immediately when the user selects a check box, we need the check box item listener to force the `paintComponent` method to be called. This is accomplished by the statement in line 82, in the `CheckBoxListener` class's `itemStateChanged` method. This statement calls the `drawingPanel` object's `repaint` method, which causes the `drawingPanel` object's surface to be cleared, and then causes the object's `paintComponent` method to execute. Because it is in the item listener, it is executed each time the user clicks on a check box.

## Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

25.18 In an AWT component, or a class that extends `JApplet` or `JFrame`, if you want to get a reference to the `Graphics` object, do you override the `paint` or `paintComponent` method?

25.19 In a `JPanel` object, do you override the `paint` or `paintComponent` method to get a reference to the `Graphics` object?

25.20 When are the `paint` and `paintComponent` method called?

25.21 In the `paint` or `paintComponent` method, what should be done before anything else?

25.22 How do you force the `paint` or `paintComponent` method to be called?

25.23 When using a `Graphics` object to draw an oval, what invisible shape is the oval enclosed in?

25.24 What values are contained in the two arrays that are passed to a `Graphics` object's `drawPolygon` method?

25.25 What `Graphics` class methods do you use to perform the following tasks?

a) Draw a line.
b) Draw a filled rectangle.
c) Draw a filled oval.
d) Draw a filled arc.
e) Set the drawing color.
f) Draw a rectangle.
g) Draw an oval.
h) Draw an arc.
i) Draw a string.
j) Set the font.

## 25.6 Handling Mouse Events

**CONCEPT: Java allows you to create listener classes that handle events generated by the mouse.**

### Handling Mouse Events

The mouse generates two types of events: mouse events and mouse motion events. To handle mouse events you create a *mouse listener* class and/or a *mouse motion listener* class. A mouse listener class can respond to any of the follow events:

- The mouse button is pressed.
- The mouse button is released.
- The mouse button is clicked (pressed, then released without moving the mouse).
- The mouse cursor enters a component's screen space.
- The mouse cursor exits a component's screen space.

A mouse listener class must implement the `MouseListener` interface, which is in the `java.awt.event` package. The class must also have the methods listed in Table 25-3.

**Table 25-3** Methods required by the `MouseListener` interface

| Method | Description |
|---|---|
| `public void mousePressed(MouseEvent e)` | If the mouse cursor is over the component and the mouse button is pressed, this method is called. |
| `public void mouseClicked(MouseEvent e)` | A mouse click is defined as pressing the mouse button and releasing it without moving the mouse. If the mouse cursor is over the component and the mouse is clicked on, this method is called. |
| `public void mouseReleased(MouseEvent e)` | This method is called when the mouse button is released after it has been pressed. The `mousePressed` method is always called before this method. |
| `public void mouseEntered(MouseEvent e)` | This method is called when the mouse cursor enters the screen area belonging to the component. |
| `public void mouseExited(MouseEvent e)` | This method is called when the mouse cursor leaves the screen area belonging to the component. |

Notice that each of the methods listed in Table 25-3 accepts a `MouseEvent` object as its argument. The `MouseEvent` object contains data about the mouse event. We will use two of the `MouseEvent` object's methods: `getX` and `getY`. These methods return the *X* and *Y* coordinates of the mouse cursor when the event occurs.

Once you create a mouse listener class, you can register it with a component using the `addMouseListener` method, which is inherited from the `Component` class. The appropriate methods in the mouse listener class are automatically called when their corresponding mouse events occur.

A mouse motion listener class can respond to the following events:

- The mouse is dragged (the button is pressed and the mouse is moved while the button is held down).
- The mouse is moved.

A mouse motion listener class must implement the `MouseMotionListener` interface, which is in the `java.awt.event` package. The class must also have the methods listed in Table 25-4. Notice that each of these methods also accepts a `MouseEvent` object as an argument.

**Table 25-4** Methods required by the `MouseMotionListener` interface

| Method | Description |
|---|---|
| `public void mouseDragged(MouseEvent e)` | The mouse is dragged when its button is pressed and the mouse is moved while the button is held down. This method is called when a dragging operation begins over the component. The `mousePressed` method is always called just before this method. |
| `public void mouseMoved(MouseEvent e)` | This method is called when the mouse cursor is over the component and it is moved. |

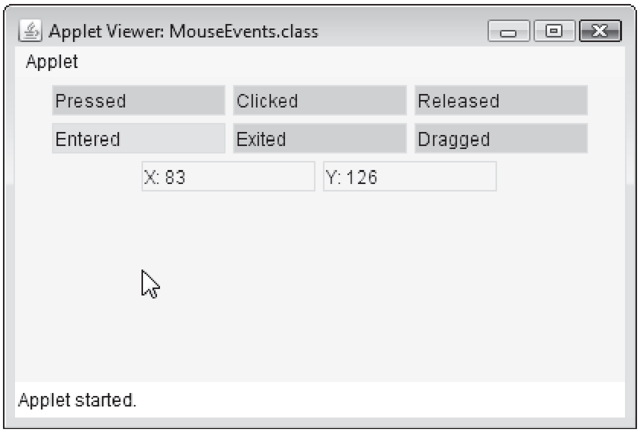
Once you create a mouse motion listener class, you can register it with a component using the `addMouseMotionListener` method, which is inherited from the `Component` class. The appropriate methods in the mouse motion listener class are automatically called when their corresponding mouse events occur.

The `MouseEvents` class, shown in Code Listing 25-19, is an applet that demonstrates both a mouse listener and a mouse motion listener. The file *MouseEvents.html*, which is in the same folder as the applet class, can be used to start the applet. Figure 25-29 shows the applet running. The window displays a group of read-only text fields that represent the different mouse and mouse motion events. When an event occurs, the corresponding text field turns yellow. The last two text fields constantly display the mouse cursor's *X* and *Y* coordinates. Run this applet and experiment by clicking the mouse inside the window, dragging the mouse, moving the mouse cursor in and out of the window, and moving the mouse cursor over the text fields.

**Code Listing 25-19** (`MouseEvents.java`)

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
   This applet shows the mouse events as they occur.
*/

public class MouseEvents extends JApplet
{
   private JTextField[] mouseStates;
   private String[] text = {
                 "Pressed", "Clicked", "Released",
                 "Entered", "Exited", "Dragged",
                 "X:", "Y:" };

   /**
      init method
   */

   public void init()
   {
      // Create a layout manager.
      setLayout(new FlowLayout());

      // Create the array of text fields.
      mouseStates = new JTextField[8];
      for (int i = 0; i < mouseStates.length; i++)
      {
         mouseStates[i] = new JTextField(text[i], 10);
         mouseStates[i].setEditable(false);
         add(mouseStates[i]);
      }

      // Add a mouse listener to this applet.
      addMouseListener(new MyMouseListener());

      // Add a mouse motion listener to this applet.
      addMouseMotionListener(new MyMouseMotionListener());
   }

   /**
      The clearTextFields method sets all of the text
      backgrounds to light gray.
   */

   public void clearTextFields()
   {
      for (int i = 0; i < 6; i++)
         mouseStates[i].setBackground(Color.lightGray);
   }

   /**
      Private inner class that handles mouse events.
      When an event occurs, the text field for that
      event is given a yellow background.
   */

   private class MyMouseListener
                      implements MouseListener
   {
      public void mousePressed(MouseEvent e)
      {
         clearTextFields();
         mouseStates[0].setBackground(Color.yellow);
      }

      public void mouseClicked(MouseEvent e)
      {
         clearTextFields();
         mouseStates[1].setBackground(Color.yellow);
      }

      public void mouseReleased(MouseEvent e)
      {
         clearTextFields();
         mouseStates[2].setBackground(Color.yellow);
      }

      public void mouseEntered(MouseEvent e)
      {
         clearTextFields();
         mouseStates[3].setBackground(Color.yellow);
      }

      public void mouseExited(MouseEvent e)
      {
         clearTextFields();
         mouseStates[4].setBackground(Color.yellow);
      }
   }

   /**
      Private inner class to handle mouse motion events.
   */

   private class MyMouseMotionListener
                        implements MouseMotionListener
   {
      public void mouseDragged(MouseEvent e)
      {
         clearTextFields();
         mouseStates[5].setBackground(Color.yellow);
      }

      public void mouseMoved(MouseEvent e)
      {
         mouseStates[6].setText("X:  " + e.getX());
         mouseStates[7].setText("Y:  " + e.getY());
      }
   }
}
```

**Figure 25-29** MouseEvents applet (Oracle Corporate Counsel)



Pressed
Clicked
Released
Entered
Exited
Dragged
X: 83
Y: 126

Applet started.

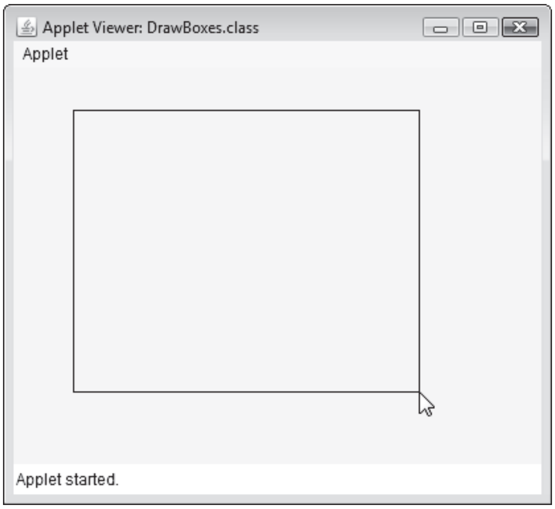
### Using Adapter Classes

Many times when you handle mouse events, you will not be interested in handling every event that the mouse generates. This is the case with the `DrawBoxes` applet, which handles only mouse pressed and mouse dragged events.

This applet lets you draw rectangles by pressing the mouse button and dragging the mouse inside the applet window. When you initially press the mouse button, the position of the mouse cursor becomes the upper-left corner of a rectangle. As you drag the mouse, the lower-right corner of the rectangle follows the mouse cursor. When you release the mouse cursor, the rectangle stops following the mouse. Figure 25-30 shows an example of the applet's window. You can run the applet with the *DrawBoxes.html* file, which is in the same folder as the applet class. Code Listing 25-20 shows the code for the `DrawBoxes` class.

**NOTE:** To draw the rectangle, you must drag the mouse cursor to the right and below the position where you initially pressed the mouse button.

**Figure 25-30** DrawBoxes applet (Oracle Corporate Counsel)



Applet started.

**Code Listing 25-20** (`DrawBoxes.java`)

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
   This applet demonstrates how mouse events and mouse
   motion events can be handled. It lets the user draw
   boxes by dragging the mouse.
*/

public class DrawBoxes extends JApplet
{
   private int currentX = 0;     // Mouse cursor's X position
   private int currentY = 0;     // Mouse cursor's Y position
   private int width = 0;        // The rectangle's width
   private int height = 0;       // The rectangle's height

   /**
      init method
   */

   public void init()
   {
      // Add a mouse listener and a mouse motion listener.
      addMouseListener(new MyMouseListener());
      addMouseMotionListener(new MyMouseMotionListener());
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass's paint method.
      super.paint(g);

      // Draw a rectangle.
      g.drawRect(currentX, currentY, width, height);
   }

   /**
      Mouse listener class
   */

   private class MyMouseListener
                     implements MouseListener
   {
      public void mousePressed(MouseEvent e)
      {
         // Get the mouse cursor coordinates.
         currentX = e.getX();
         currentY = e.getY();
      }

      //
      // The following methods are unused, but still
      // required by the MouseListener interface.
      //

      public void mouseClicked(MouseEvent e)
      {
      }

      public void mouseReleased(MouseEvent e)
      {
      }

      public void mouseEntered(MouseEvent e)
      {
      }

      public void mouseExited(MouseEvent e)
      {
      }
   }

   /**
      Mouse Motion listener class
   */

   private class MyMouseMotionListener
                     implements MouseMotionListener
   {
      public void mouseDragged(MouseEvent e)
      {
         // Calculate the size of the rectangle.
         width = e.getX() - currentX;
         height = e.getY() - currentY;

         // Repaint the window.
         repaint();
      }

      /**
         The mouseMoved method is unused, but still
         required by the MouseMotionListener interface.
      */

      public void mouseMoved(MouseEvent e)
      {
      }
   }
}
```

Notice in the mouse listener and mouse motion listener classes that several of the methods are empty. Even though the applet handles only two mouse events, the `MyMouseListener` and `MyMouseMotionListener` classes must have all of the methods required by the interfaces they implement. If any of these methods are omitted, a compiler error results.

The Java API provides an alternative technique for creating these listener classes, which eliminates the need to define empty methods for the events you are not interested in. Instead of implementing the `MouseListener` or `MouseMotionListener` interfaces, you can extend your classes from the `MouseAdapter` or `MouseMotionAdapter` classes. These classes implement the `MouseListener` and `MouseMotionListener` interfaces and provide empty definitions for all of the required methods. When you extend a class from one of these adapter classes, it inherits the empty methods. In your extended class, you can override the methods you want and forget about the others. Both the `MouseAdapter` and `MouseMotionAdapter` classes are in the `java.awt.event` package.

The `DrawBoxes2` class shown in Code Listing 25-21 is a modification of the `DrawBoxes` class previously shown. In this version, the `MyMouseListener` class extends `MouseAdapter` and the `MyMouseMotionListener` class extends `MouseMotionAdapter`. This applet operates exactly the same as the `DrawBoxes` applet. The only difference is that this class does not have the empty methods in the listener classes.

**NOTE:** Java provides an adapter class for all of the interfaces in the API that have more than one method.

**Code Listing 25-21** (`DrawBoxes2.java`)

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
   This applet demonstrates how the mouse adapter
   classes can be used.
*/

public class DrawBoxes2 extends JApplet
{
   private int currentX = 0;      // Mouse cursor's X position
   private int currentY = 0;      // Mouse cursor's Y position
   private int width = 0;         // The rectangle's width
   private int height = 0;        // The rectangle's height

   /**
      init method
   */

   public void init()
   {
      // Add a mouse listener and a mouse motion listener.
      addMouseListener(new MyMouseListener());
      addMouseMotionListener(new MyMouseMotionListener());
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass's paint method.
      super.paint(g);

      // Draw a rectangle.
      g.drawRect(currentX, currentY, width, height);
   }

   /**
      Mouse listener class
   */

   private class MyMouseListener extends MouseAdapter
   {
      public void mousePressed(MouseEvent e)
      {
         // Get the coordinates of the mouse cursor.
         currentX = e.getX();
         currentY = e.getY();
      }
   }

   /**
      Mouse Motion listener class
   */

   private class MyMouseMotionListener
                           extends MouseMotionAdapter
   {
      public void mouseDragged(MouseEvent e)
      {
         // Calculate the size of the rectangle.
         width = e.getX() - currentX;
         height = e.getY() - currentY;

         // Repaint the window.
         repaint();
      }
   }
}
```

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

25.26 What is the difference between a mouse press event and a mouse click event?

25.27 What interface would a listener class implement to handle a mouse click event? A mouse press event? A mouse dragged event? A mouse release event? A mouse move event?

25.28 What type of object do mouse listener and mouse motion listener methods accept? What methods do these types of objects provide for determining a mouse cursor's location?

25.29 If a class implements the `MouseListener` interface but does not need to use all of the methods specified by the interface, can the definitions for those methods be left out? If not, how are these methods dealt with?

25.30 What is an adapter class, and how does it make some programming tasks easier?

## 25.7 Timer Objects

**CONCEPT: A `Timer` object regularly generates action events at programmer-specified time intervals.**

`Timer` objects automatically generate action events at regular time intervals. This is useful when you want a program to perform an operation at certain times or after an amount of time has passed.

`Timer` objects are created from the `Timer` class, which is in the `javax.swing` package. Here is the general format of the `Timer` class's constructor:

```
Timer(int delay, ActionListener listener)
```

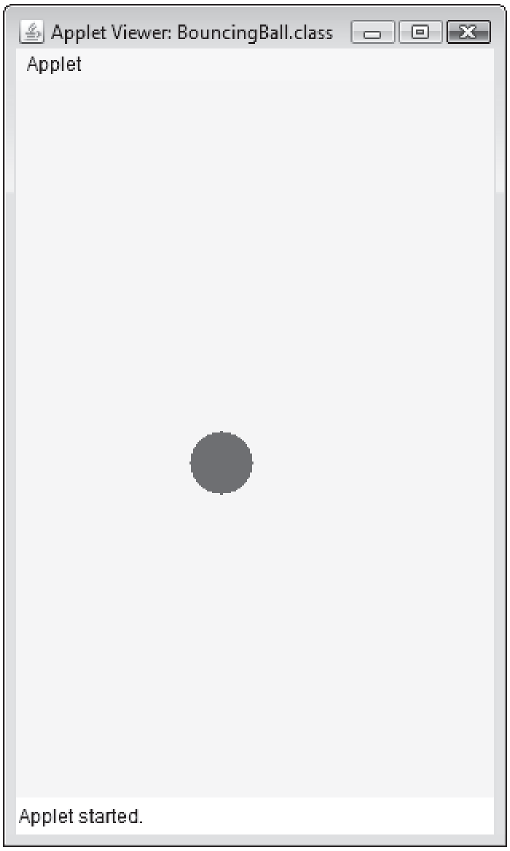
The argument passed into the *`delay`* parameter is the amount of time between action events, measured in milliseconds. A millisecond is a thousandth of a second, so a *`delay`* value of 1000 causes an action event to be generated every second. The argument passed into the *`listener`* parameter is a reference to an action listener that is to be registered with the `Timer` object. If you want to add an action listener at a later time, you can pass `null` as this argument, then use the `Timer` object's `addActionListener` method to register an action listener. Table 25-5 lists the `Timer` class's methods.

An application can use a `Timer` object to execute code automatically at regular time intervals. For example, a `Timer` object can be used to perform simple animation by moving a graphic image across the screen by a certain amount at regular time intervals. This is demonstrated in the `BouncingBall` class, shown in Code Listing 25-22. This class is an applet that displays a bouncing ball, as shown in Figure 25-31.

**Table 25-5** Timer class methods

| Method | Description |
|---|---|
| `void addActionListener (ActionListener listener)` | Registers the object referenced by *`listener`* as an action listener. |
| `int getDelay()` | Returns the current time delay in milliseconds. |
| `Boolean isRunning()` | Returns `true` if the `Timer` object is running. Otherwise, it returns `false`. |
| `void setDelay(int delay)` | Sets the time delay. The argument is the amount of the delay in milliseconds. |
| `void start()` | Starts the `Timer` object, which causes it to generate action events. |
| `void stop()` | Stops the `Timer` object, which causes it to stop generating action events. |

**Figure 25-31** `BouncingBall` applet (Oracle Corporate Counsel)



Applet started.

**Code Listing 25-22** (BouncingBall.java)

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

/**
   This applet uses a Timer object to animate
   a bouncing ball.
*/

public class BouncingBall extends JApplet
{
   private final int X = 109;             // Ball's X coordinate
   private final int WIDTH = 40;          // Ball's width
   private final int HEIGHT = 40;         // Ball's height
   private final int TIME_DELAY = 30;     // Time delay
   private final int MOVE = 20;           // Pixels to move ball
   private final int MINIMUM_Y = 50;      // Min height of ball
   private final int MAXIMUM_Y = 400;     // Max height of ball
   private int y = 400;                   // Ball's Y coordinate
   private boolean goingUp = true;        // Direction indicator
   private Timer timer;                   // Timer object


   /**
      init method
   */

   public void init()
   {
      timer = new Timer(TIME_DELAY, new TimerListener());
      timer.start();
   }

   /**
      paint method
      @param g The applet's Graphics object.
   */

   public void paint(Graphics g)
   {
      // Call the superclass paint method.
      super.paint(g);

      // Set the drawing color to red.
      g.setColor(Color.red);

47          // Draw the ball.
48          g.fillOval(X, y, WIDTH, HEIGHT);
49      }
50
51      /**
52          Private inner class that handles the Timer object's
53          action events.
54      */
55
56      private class TimerListener implements ActionListener
57      {
58         public void actionPerformed(ActionEvent e)
59         {
60            // Update the ball's Y coordinate.
61            if (goingUp)
62            {
63               if (y > MINIMUM_Y)
64                  y -= MOVE;
65               else
66                  goingUp = false;
67            }
68            else
69            {
70               if (y < MAXIMUM_Y)
71                  y += MOVE;
72               else
73                  goingUp = true;
74            }
75
76            // Force a call to the paint method.
77            repaint();
78         }
79      }
80   }
```

The `BouncingBall` class's `init` method creates a `Timer` object with the following statement in line 30:

```
timer = new Timer(TIME_DELAY, new TimerListener());
```

This initializes the object with a time delay of 30 milliseconds (the value of `TIME_DELAY`) and registers an instance of the `TimerListener` class as an action listener. This means that once the object is started, every 30 milliseconds it generates an action event, causing the action listener's `actionPerformed` method to execute. The next statement in the `init` method, in line 31, starts the `Timer` object as follows:

```
timer.start();
```

This causes the `Timer` object to commence generating action events. The `TimerListener` class's `actionPerformed` method calculates the new position of the bouncing ball and repaints the screen.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

25.31 What type of events do `Timer` objects generate?

25.32 How are the time intervals between a `Timer` object's action events measured?

25.33 How do you cause a `Timer` object to begin generating action events?

25.34 How to you cause a `Timer` object to cease generating action events?

## 25.8 Playing Audio

**CONCEPT: Sounds that have been stored in an audio file may be played from a Java program.**

Java applets can play audio that is stored in a variety of popular sound file formats. The file formats directly supported are as follows:

- *.aif* or *.aiff* (Macintosh Audio File)
- *.au* (Sun Audio File)
- *.mid* or *.rmi* (MIDI File)
- *.wav* (Windows Wave File)

To play audio files, your computer must be equipped with a sound card and speakers. One way to play an audio file is to use the `play` method, which the `JApplet` class inherits from the `Applet` class. The version of the method that we will use is as follows:

```
void play(URL baseLocation, String fileName)
```

The argument passed to *`baseLocation`* is a `URL` object that specifies the location of the file. The argument passed to *`fileName`* is the name of the file. The sound that is recorded in the file is played one time.

When calling the `play` method, it is common to use either the `getDocumentBase` or `getCodeBase` method (both of which the `JApplet` class inherits from the `Applet` class) to get a `URL` object for the first argument. The `getDocumentBase` method returns a `URL` object containing the location of the HTML file that invoked the applet. Here is an example of a call to the `play` method, using a call to `getDocumentBase` for the first argument:

```
play(getDocumentBase(), "mysound.wav");
```

This statement will load and play the *mysound.wav* sound file, stored at the same location as the HTML file that invoked the applet.

The `getCodeBase` method returns a URL object containing the location of the applet's *.class* file. Here is an example of its use:

```
play(getCodeBase(), "mysound.wav");
```

This statement will load and play the *mysound.wav* sound file, stored at the same location as the applet's *.class* file. The *AudioDemo1* folder contains an example applet that plays a sound file using the `play` method.

**NOTE:** If the sound file specified by the arguments to the play method cannot be found, no sound will be played.

## Using an `AudioClip` Object

The `Applet` class's `play` method loads a sound file, plays it one time, and then releases it for garbage collection. If you need to load a sound file to be played multiple times, you should use an *`AudioClip` object*.

An `AudioClip` object is an object that implements the `AudioClip` interface. The `AudioClip` interface is in the `java.applet` package, and it specifies the following three methods: `play`, `loop`, and `stop`. The `play` method plays a sound one time. The `loop` method repeatedly plays a sound, and the `stop` method causes a sound to stop playing.

The `Applet` class's `getAudioClip` method can be used to create an `AudioClip` object for a given sound file as follows:

```
AudioClip getAudioClip(URL baseLocation, String fileName)
```

The argument passed to *`baseLocation`* is a `URL` object that specifies the location of a sound file, and the argument passed to *`fileName`* is the name of the file. The method returns an `AudioClip` object that can be used to play the sound file.

As before, we can use the `getDocumentBase` or `getCodeBase` method to get a `URL` object for the first argument. Here is an example of a statement that uses the `getAudioClip` method:

```
AudioClip clip = getAudioClip(getDocumentBase(), "mysound.wav");
```

This statement declares `clip` as an `AudioClip` reference variable. The object returned by the `getAudioClip` method will load the *mysound.wav* file, stored at the same location as the HTML file that invoked the applet. The address of the object will be assigned to `clip`. The following statement can then be used to play the sound file:

```
clip.play();
```

The sound file can be played repeatedly with the following statement:

```
clip.loop();
```

Any time the sound file is being played, the following statement can be used to stop it:

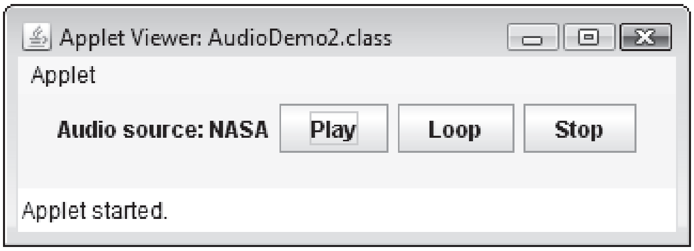
```
clip.stop();
```

The `AudioDemo2` class shown in Code Listing 25-23 is an applet that uses an `AudioClip` object to play a sound file. The file *AudioDemo2.html* can be used to start the applet. Figure 25-32 shows the applet running. The Play button calls the `AudioClip` object's `play` method, causing the sound file to play once. The Loop button calls the `loop` method, causing the sound file to be played repeatedly. The Stop button `stops` the sound file from playing. The sound file that is played is a famous NASA transmission from the Moon. NASA provides a wealth of public domain audio, video, and image files. You can find such items by going to www.nasa.gov, and then search the site using search terms such as “audio clips”, “video clips”, etc.

**Code Listing 25-23** (`AudioDemo2.java`)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import javax.swing.*;

/**
   This applet uses the AudioClip class to play a
   sound. Sound source: NASA
*/

public class AudioDemo2 extends JApplet
{
   private JLabel credit;         // Displays NASA credit
   private JButton playButton;    // Plays the sound clip
   private JButton loopButton;    // Loops the clip
   private JButton stopButton;    // Stops the clip
   private AudioClip sound;       // Holds the sound clip

   /**
      init method
   */

   public void init()
   {
      // Create a layout manager.
      setLayout(new FlowLayout());

      // Make the credit label and add it.
      credit = new JLabel("Audio source: NASA");
      add(credit);

      // Make the buttons and add them.
      makeButtons();

      // Get an AudioClip object for the sound file.
      sound = getAudioClip(getDocumentBase(), "step.wav");
   }

   /**
      The makeButtons method creates the Play, Loop, and
      Stop buttons, and adds them to the content pane.
   */

   private void makeButtons()
   {
      // Create the Play, Loop, and Stop buttons.
      playButton = new JButton("Play");
      loopButton = new JButton("Loop");
      stopButton = new JButton("Stop");

      // Register an action listener with each button.
      playButton.addActionListener(new ButtonListener());
      loopButton.addActionListener(new ButtonListener());
      stopButton.addActionListener(new ButtonListener());

      // Add the buttons to the content pane.
      add(playButton);
      add(loopButton);
      add(stopButton);
   }

   /**
      Private inner class that handles the action event
      that is generated when the user clicks one of the
      buttons.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Determine which button was clicked and
         // perform the selected action.
         if (e.getSource() == playButton)
            sound.play();
         else if (e.getSource() == loopButton)
            sound.loop();
         else if (e.getSource() == stopButton)
            sound.stop();
      }
   }
}
```

**Figure 25-32** AudioDemo2 applet (Oracle Corporate Counsel)



Applet started.

## Playing Audio in an Application

The previous examples show how to play an audio file in an applet. You can play audio in an application as well. The process of getting a reference to an `AudioClip` object is different, however, in a class that does not extend `JApplet`. In the *Chapter 19\AudioDemo3* source code folder you will find a Swing application named *AudioFrame.java* that demonstrates how to do it. The following code segment is from the application.

```
43      // Create a file object for the step.wav file.
44      File file = new File("step.wav");
45
46      // Get a URI object for the audio file.
47      URI uri = file.toURI();
48
49      // Get a URL for the audio file.
50      URL url = uri.toURL();
51
52      // Get an AudioClip object for the sound
53      // file using the Applet class's static
54      // newAudioClip method.
55      sound = Applet.newAudioClip(url);
```

In line 44, we create a `File` object representing the audio file. Then, in line 47 we call the File class's `toURI` method to create a `URI` object representing the audio file. The `URI` class is in the `java.net` package. (URI stands for Uniform Resource Identifier.)

Then, in line 50, we call the `URI` class's `toURL` method to create a `URL` object representing the audio file. Note that if this method cannot construct a `URL` it throws a checked exception—`MalformedURLException`. The `MalformedURLException` class is in the `java.net` package.

Last, in line 55, we call the `Applet` class's static `newAudioClip` method, passing the `URL` object as an argument. The method returns a reference to an `AudioClip` object which can be used as previously demonstrated to play the audio file.

**Checkpoint**

MyProgrammingLab™ *www.myprogramminglab.com*

25.35 What `Applet` method can you use to play a sound file?

25.36 What is the difference between using the `Applet` method asked for in Checkpoint 25.35, and using an `AudioClip` object to play a sound file?

25.37 What methods does an `AudioClip` object have? What do they do?

25.38 What is the difference between the `Applet` class's `getDocumentBase` and `getCodeBase` methods?

## 25.9 Common Errors to Avoid

- **Forgetting a closing tag in an HTML document.** Most HTML tags have an opening tag and a closing tag. The page will not appear properly if you forget a closing tag.
- **Confusing the `<head></head>` tag with `<h1></h1>` or another header tag.** The `<head></head>` tag marks a document's head section, whereas the `<h1></h1>` tag marks a header, which is large bold text.
- **Using *X* and/or *Y* coordinates that are outside of the component when drawing a shape.** If you use coordinates that are outside the component to draw a shape, the shape will not appear.
- **Not calling the superclass's `paint` or `paintComponent` method.** When you override the `paint` or `paintComponent` method, the overriding method should call the superclass's version of the method before doing anything else.
- **Overriding the `paint` method with a component extended from `JComponent`.** You should override the `paint` method only with AWT components, `JFrame` components, or `JApplet` components.
- **Not calling the `repaint` method to redisplay a window.** When you update the data used to draw shapes on a component, you must call the `repaint` method to force a call to the `paint` or `paintComponent` method.
- **Not providing empty definitions for the unneeded methods in a mouse listener or mouse motion listener class.** When writing mouse listeners or mouse motion listeners, you must provide definitions for all the methods specified by the listener interfaces. To avoid this you can write a listener as a class that inherits from an adapter class.
- **Forgetting to start a `Timer` object.** A `Timer` object does not begin generating action events until it is started with a call to its `start` method.

## Review Questions and Exercises

### Multiple Choice and True/False

1. This section of an HTML document contains all of the tags and text that produce output in the browser window.
   a. head
   b. content
   c. body
   d. output

2. You place the `<title></title>` tag in this section of an HTML document.
   a. head
   b. content
   c. body
   d. output
3. Everything that appears between these tags in an HTML document is the content of the Web page.
   a. `<content></content>`
   b. `<html></html>`
   c. `<head></head>`
   d. `<page></page>`
4. To create a level one header you use this tag.
   a. `<level1></level1>`
   b. `<header1></header1>`
   c. `<h1></h1>`
   d. `<head></head>`
5. When using Swing to write an applet, you extend the applet's class from this class.
   a. `Applet`
   b. `JApplet`
   c. `JFrame`
   d. `JAppletFrame`
6. When using AWT to write an applet, you extend the applet's class from this class.
   a. `Applet`
   b. `JApplet`
   c. `JFrame`
   d. `JAppletFrame`
7. This applet method is invoked instead of a constructor.
   a. `startUp`
   b. `beginApplet`
   c. `invoke`
   d. `init`
8. The Sun JDK comes with this program, which loads and executes an applet without the need for a Web browser.
   a. `applettest`
   b. `appletload`
   c. `appletviewer`
   d. `viewapplet`
9. A class that inherits from `Applet` or `Frame` does not have one of these.
   a. an `add` method
   b. an `init` method
   c. a content pane
   d. a layout manager

10. What location on a component usually has the coordinates (0, 0)?
    a. upper-right corner
    b. upper-left corner
    c. center
    d. lower-right corner
11. In a class that extends `JApplet` or `JFrame` you override this method to get a reference to the `Graphics` object.
    a. `paint`
    b. `paintComponent`
    c. `getGraphics`
    d. `graphics`
12. In a class that extends `JPanel` you override this method to get a reference to the `Graphics` object.
    a. `paint`
    b. `paintComponent`
    c. `getGraphics`
    d. `graphics`
13. The `drawLine` method is a member of this class.
    a. `JApplet`
    b. `Applet`
    c. `JFrame`
    d. `Graphics`
14. To force the `paint` method to be called to update a component's display, you __________.
    a. call the `paint` method
    b. call the `repaint` method
    c. call the `paintAgain` method
    d. do nothing; you cannot force the `paint` method to be called
15. A class that implements this interface can handle mouse dragged events.
    a. `MouseListener`
    b. `ActionListener`
    c. `MouseMotionListener`
    d. `MouseDragListener`
16. A class that implements this interface can handle mouse click events.
    a. `MouseListener`
    b. `ActionListener`
    c. `MouseMotionListener`
    d. `MouseDragListener`
17. This `MouseEvent` method returns the *X* coordinate of the mouse cursor at the moment the mouse event is generated.
    a. `getXCoord`
    b. `getMouseX`
    c. `getPosition`
    d. `getX`

18. If a class implements a standard API interface that specifies more than one method but does not need many of the methods, this should be used instead of the interface.
    a. your own detailed versions of the needed methods
    b. an adapter class
    c. a different interface
    d. there is no other choice
19. A `Timer` object's time delay between events is specified in this unit of time.
    a. seconds
    b. microseconds
    c. milliseconds
    d. minutes
20. A `Timer` object generates this type of event.
    a. action events
    b. timer events
    c. item events
    d. interval events
21. The following `Applet` class method returns a `URL` object with the location of the HTML file that invoked the applet.
    a. `getHTMLlocation`
    b. `getDocumentBase`
    c. `getAppletBase`
    d. `getCodeBase`
22. The following `Applet` class method returns a `URL` object with the location of the applet's *.class* file.
    a. `getHTMLlocation`
    b. `getDocumentBase`
    c. `getAppletBase`
    d. `getCodeBase`
23. **True or False:** Applets cannot create files on the user's system.
24. **True or False:** Applets can read files on the user's system.
25. **True or False:** Applets cannot make network connections with any system except the server from which the applet was transmitted.
26. **True or False:** Applets can retrieve information about the user's system or the user's identity.
27. **True or False:** The `<h6>` tag produces larger text than the `<h1>` tag.
28. **True or False:** You use a static `main` method to create an instance of an applet class.
29. **True or False:** In a class that extends `JApplet`, you add components to the content pane.
30. **True or False:** In an applet, events are handled differently than in a GUI application.
31. **True or False:** An object of the `Frame` class does not have a content pane.
32. **True or False:** In an overriding `paint` method, you should never call the superclass's version of the `paint` method.

33. **True or False:** Once a `Timer` object has been started, it cannot be stopped without shutting down the program.
34. **True or False:** The `Applet` class's play method loads and plays an audio file once and then releases the memory it occupies for garbage collection.
35. **True or False:** The `loop` and `stop` methods, for use with audio files, are part of the `Applet` class.

### Find the Error

Find the errors in the following code:

1.
   ```
   <applet code="MyApplet.java" width=100 height=50>
   </applet>
   ```
2.
   ```
   public void paint(Graphics g)
   {
      drawLine(0, 0, 100, 100);
   }
   ```
3.
   ```
   // Force a call to the paint method.
   paint();
   ```
4.
   ```
   public class MyPanel extends JPanel
   {
      public MyPanel()
      {
         // Constructor code...
      }
      public void paint(Graphics g)
      {
         //paint method code...
      {
   }
   ```
5.
   ```
   private class MyMouseListener implements MouseListener
   {
      public void mouseClicked(MouseEvent e)
      {
         mouseClicks += 1;
      }
   }
   ```
6.
   ```
   private class MyMouseListener implements MouseAdapter
   {
      public void mouseClicked(MouseEvent e)
      {
         mouseClicks += 1;
      }
   }
   ```

## Algorithm Workbench

1. Write the text and HTML tags necessary to display “My Home Page” as a level one header, centered in the browser window.
2. You have written an applet and saved the source code in a file named `MyApplet.java`. Write the HTML tag needed to execute the applet in an area that is 300 pixels wide by 200 pixels high. Assume that the compiled applet code is stored in the same directory as the HTML document.
3. Look at the following GUI application class and indicate by line number the changes that should be made to convert this to an applet using Swing:

```
 1 public class SimpleWindow extends JFrame
 2 {
 3    public SimpleWindow()
 4    {
 5       // Set the title.
 6       setTitle("A Simple Window");
 7
 8       // Specify what happens when the close button is clicked.
 9       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10
11       // Add a label.
12       JLabel label = new JLabel("This is a simple window.");
13       add(label);
14
15       // Pack and display the window.
16       pack();
17       setVisible(true);
18    }
19 }
```

4. Assume that `g` references a `Graphics` object. Write code that performs the following:
   - a. Draws an outline of a rectangle that is 100 pixels wide by 200 pixels high, with its upper-left corner at (50, 75).
   - b. Draws a filled rectangle that is 300 pixels wide by 100 pixels high, with its upper-left corner at (10, 90).
   - c. Draws a blue outline of an oval with an enclosing rectangle that is 100 pixels wide by 50 pixels high, with its upper-left corner at (10, 25).
   - d. Draws a red line from (0, 5) to (150, 175).
   - e. Draws the string “Greetings Earthling”. The lower-left point of the string should be at (80, 99). Use a bold, 20-point serif font.
   - f. Draws a polygon with vertices at the following points: (10, 10), (10, 25), (50, 25), and (50, 10). What shape does this code result in?

5. Rewrite the following mouse motion listener so it uses an adapter class:

```
private class MyMouseMotionListener implements MouseMotionListener
{
   public void mouseDragged(MouseEvent e)
   {
   }
   public void mouseMoved(MouseEvent e)
   {
      mouseMovements += 1;
   }
}
```

6. Assume that a class has an inner class named `MyTimerListener` that can be used to handle the events generated by a `Timer` object. Write code that creates a `Timer` object with a time delay of one half second. Register an instance of `MyTimerListener` with the class.

### Short Answer

1. When a user accesses a Web page on a remote server with his or her browser, and that Web page has an applet associated with it, is the applet executed by the server or by the user's system?
2. List at least three security restrictions imposed on applets.
3. Why are applets sometimes necessary in Web page development?
4. Why isn't it necessary to call the `setVisible` method to display an applet?
5. Why would you ever need to use the older AWT library instead of Swing to develop an applet?
6. A panel is 600 pixels wide by 400 pixels high. What are the *X* and *Y* coordinates of the pixel in the upper-left corner? The upper-right corner? The lower-left corner? The lower-right corner? The center of the panel?
7. When is a component's `paint` or `paintComponent` method called?
8. What is an adapter class? How does it make some programming tasks more convenient? Under what circumstances does the Java API provide an adapter class?
9. Under what circumstances would you want to use an `AudioClip` object to play a sound file, rather than the `Applet` class's play method?

## Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

### 1. `FollowMe` Applet

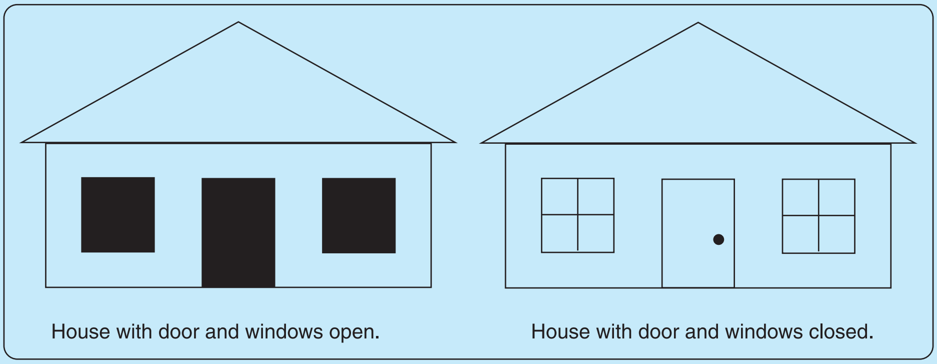
Write an applet that initially displays the word "Hello" in the center of a window. The word should follow the mouse cursor when it is moved inside the window.

VideoNote
The House Applet Problem

### 2. House Applet

Write an applet that draws the house shown on the left in Figure 25-33. When the user clicks on the door or windows, they should close. The figure on the right shows the house with its door and windows closed.

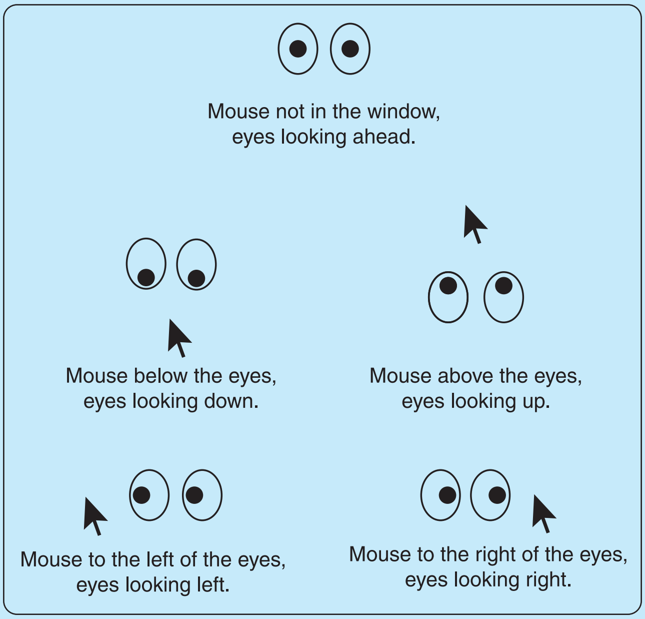
**Figure 25-33** House drawing


House with door and windows open.
House with door and windows closed.

### 3. `WatchMe` Applet

Write an applet that displays a drawing of two eyes in the center of its window. When the mouse cursor is not inside the window, the eyes should look ahead. When the mouse cursor is inside the window, the eyes should follow the cursor. This is illustrated in Figure 25-34.

**Figure 25-34** Eyes following the mouse cursor


Mouse not in the window,
eyes looking ahead.
Mouse below the eyes,
eyes looking down.
Mouse above the eyes,
eyes looking up.
Mouse to the left of the eyes,
eyes looking left.
Mouse to the right of the eyes,
eyes looking right.

### 4. `Thermometer` Applet

Write an applet that displays a thermometer. The user should be able to control the temperature with a slider component. When the user moves the slider, the thermometer should show the corresponding temperature.

### 5. Polygon Drawer

Write an applet that lets the user click on six points. After the sixth point is clicked, the applet should draw a polygon with a vertex at each point the user clicked.

### 6. `GridFiller` Applet

Write an applet that displays a 4 × 4 grid. When the user clicks on a square in the grid, the applet should draw a filled circle in it. If the square already has a circle, clicking on it should cause the circle to disappear.

### 7. `DrinkMachine` Applet

Write an applet that simulates a soft drink vending machine. The simulated machine dispenses the following soft drinks: cola, lemon-lime soda, grape soda, root beer, and bottled water. These drinks cost $0.75 each to purchase.

When the applet starts, the drink machine should have a supply of 20 of each of the drinks. The applet should have a text field where the user can enter the amount of money he or she is giving the machine. The user can then click on a button to select a drink to dispense. The applet should also display the amount of change it is giving back to the user. The applet should keep track of its inventory of drinks and inform the user whether he or she has selected a drink that is out of stock. Be sure to handle operator errors such as selecting a drink with no money entered and selecting a drink with an inadequate amount of money entered.

### 8. `Stopwatch` Applet

Write an applet that simulates a stopwatch. It should have a Start button and a Stop button. When the Start button is clicked the applet should count the seconds that pass. When the Stop button is clicked, the applet should stop counting seconds.

### 9. Slideshow Application

Write an application that displays a slideshow of images, one after the other, with a time delay between each image. The user should be able to select up to 10 images for the slide show and specify the time delay in seconds.

# Index

**A**

abstract classes and methods, 660–667
access
  package, 646–647
  sequential file, 739
  specification in UML diagrams, 339
accessing
  binary search algorithm, recursive, 1014–1017
accessing variables, within lambda expressions, 688
accessor method, 338
access specifiers
  private, 326
  protected, 641–646
  public, 28, 326
accumulators, 217, 219, 427
actual parameters, 281
adding/inserting items in `ArrayList`, 473, 477–478
addition operator, 55
address, 4
adjacency list, 1235–1237
aggregation
  description of, 515–525
  security issues, 523–525
  in UML diagrams, 523
algorithms, 6, 1070–1072
  analysis, 1063–1072
  array, 426–433
  binary search, 466–468
  binary search, recursive, 1014–1017
  selection sort, 463–466
  sequential search, 447–449
ALU (arithmetic and logic unit), 3
analysis of algorithms, 1063–1072
  asymptotic complexity, 1068–1069
  average case complexity, 1067
  basic steps, 1064
  Big O notation, 1068–1070
  complexity of algorithms, 1065–1066
  computational problems, 1064–1065
  constant time, 1069
  instances, 1064
  linear time, 1069
  logarithmic time, 1069
  n log n time, 1069
  polynomial time, 1070
  quadratic time, 1069
  space criterion, 1063
  time criterion, 1063
  worst case complexity, 1066–1067
AND (`&&`), 134–137
animation, 940–941
  controlling, 956
  `FadeTransition` class, 954–955
  `FillTransition` class, 953–954
  `RotateTransition` class, 944–948
  `ScaleTransition` class, 949–951
  `StrokeTransition` class, 952–953
`Animation` class, methods inherited from, 956
anonymous inner classes, 681–684
  and event handlers, 803–805
  and key event handler, 980–981
anonymous subclasses, 1171
API (application programmer interface), 33
  standard packages, 381
`append()`, 581–582
appending data to files, 236–237, 738
`Application` class, JavaFX, 763–765
applications
  software, 5
`Arc` class, 924–928
`ArcType` class, 925–926
`ArgumentList`, 172
arguments, 34
  arrays as, 422–424
  command-line, 468–469
  formatting string, 170–171
  object references/arguments to methods, passing, 285–88
  objects as, passing, 358–360, 500–503
  other names for, 281
  parameter data type compatibility and, 281–282
  passing multiple, 282–283
  passing, to a method, 279–289
  variable length argument lists, 470–472
arithmetic and logic unit (ALU), 3
arithmetic operators
  `+` (addition), 55
  associativity of, 58
  `/` (division), 55, 56
  `%` (modulus), 55, 56

arithmetic operators *(continued)*
  `*` (multiplication), *55*
  –(negation), *55*
  precedence of operations, *57–59*
  `+` (string concatenation), 40–41
  –(subtraction), *55*
`ArrayList` class, 1079–1081, 1127, 1135–1137
  adding/inserting items, 473, 477–478
  autoboxing and unboxing, 598–599
  capacity, 479–480
  constructor, 479, 480
  defined, 472
  diamond operator for type inference, 480
  enhanced `for` loop with, 474–475, 1131
  `get()`, 473–474
  interface variable to reference, 1141–1142
  list creation and initialization, 1136–1137
  object, creating and using, 473–474
  removing items, 476–478
  set/replacing items, 478–480
  `size()`, 473
  `toString()`, 475–476
arrays
  accessing elements of, 405–406
  algorithms, 426–433, 463–468
  as arguments, 422–426
  averaging values in numeric, 428
  binary search algorithm, 466–468
  bounds checking, 409–410
  bubble sort, 1027–1036
  comparing, 426–427
  content, inputting/outputting, 406–409
  converting `ObservableList` to, 867
  declaration notation, 412
  defined, 403
  elements, displaying, 413–415, 457
  enhanced `for` loop, 416–417
  errors, off-by-one, 410, 416
  files and, 438–439
  finding highest/lowest values in numeric, 428–429
  initialization, 411–412, 454, 456
  initializing `ListView` control with, 861–862
  insertion sort, 1027, 1042–1047
  invalid subscripts with, 409–410
  `length` field in, 416, 444, 455–456
  length of, 415–416
  of objects, 433–436, 444–447
  one-dimensional, 450
  partially filled, 437–438
  passing, to methods, 422–426, 459–461
  Quicksort, 1027, 1048–1052
  ragged, 461
  reference variables, reassigning, 419–420
  returning, from methods, 439–441
  search algorithms, 1052–1062
  selection sort algorithm, 463–466, 1027, 1036–1042
  sequential search algorithm, 447–449, 1052–1055, 1063
  size declarator, 404–406
  size, user specified, 417–419
  sorting algorithms, 1027–1052
  `String`, 441–444
  subscripts, 405, 409–410, 449
  summing values in numeric, 427–429, 457–459
  with three or more dimensions, 462
  two-dimensional, 450–461
`Arrays` class, 1172
ascending order, 1027
aspect ratio, images, 775–776
assignment operators, 52
  combined, 63–64
assignment statements, 39, 52–53
associativity, 139–140
  of arithmetic operators, 58
`@exception`, 731–732
`@Override annotation`, 635
`@param`, 288–289
`@return`, 296–299
attributes, 20
autoboxing, 595–596, 1087
average case complexity function, 1067
AVL trees, 1315–1328
  adding elements to, 1316–1319
  double LR rotation, 1316, 1317, 1319
  double LR rotation at N, 1316
  double RL rotation, 1316, 1319
  double RL rotation at M, 1316
  height, 1315
  imbalance, 1316
  implementation of, 1319–1328
  LL imbalance, 1316, 1317
  LR imbalance, 1316, 1317, 1319
  RL imbalance, 1316, 1319
  RR imbalance, 1316, 1318
  single left rotation at N, 1316, 1318
  single right rotation at N, 1316, 1318

**B**

backslashes, 36–38, 237
backspace, 37
backups, external hard drives for, *5*
`BankAccount`, 372–378
base case, 1006
base class, 612
BASIC, 7
Bell Laboratories, 7
bi-function, 1126
Big O notation, 1068–1070
binary digit, 4
binary files
  appending data to, 738
  `DataInputStream` class, 736–738
  `DataOutputStream` class, 733–735
  defined, 230, 732–733
  `FileInputStream` class, 736–737
  `FileOutputStream` class, 733–735, 738
  raw format, 733
  reading data from, 736–738
  writing data to, 733–735
binary numbers, 6
binary operators, 1126
binary search algorithm, 466–468, 1067
  Big O notation, 1069
  efficiency of, 1059
  pseudocode, 1056, 1060
  recursive, 1014–1017, 1059–1062

  Towers of Hanoi, 1017–1021
  worst case complexity of, 1066
binary search trees, 1297–1315
  adding a value to, 1299
  defined, 1298
  graphical display, 1308–1310
  implementation, 1303–1308
  removing a value from, 1299–1303
binary trees, 1287–1345
  ancestor, 1289
  applications, 1289
  array, storing in, 1335–1336
  AVL trees, 1315–1328
  binary search trees, 1297–1315
  complete, 1334–1335
  depth of, 1335
  descendant, 1288
  expressions, representing, 1289
  graphical display, 1293–1296
  heap, 1335
  heap order property, 1333–1334
  height, 1315
  inorder traversal, 1290, 1291
  leaf, 1287
  left child, 1287
  left subtree, 1288
  node, 1289–1290
  parent, 1287
  postorder traversal, 1290
  predecessor, 1287
  preorder traversal, 1290
  right child, 1287
  right subtree, 1288
  root, 1287
  subtree, 1288–1289
  traversing, 1290–1293
  visiting a node, 1291
binding
  defined, 371
  dynamic, 656–658
  late, 657
bit, 4
blank lines, 28
block comments, 80
block of statements, 118, 197
body
  class, 29
  loop, 194
  method, 30, 271
`boolean` data type, 50
`boolean` expressions
  logical operators and, 134, 135
  relational operators and, 114
`boolean` value, returning, 300
`BorderPane` layout container, 808–812
bounds checking, 409–411
`BoxBlur` class, 958, 963
braces, 29–31, 197
`break` statements, 152–153, 229
breadth first search, 1278–1281
bubble sort, 1027–1036
  Big O notation, 1070
  pseudocode, 1030
  sorting objects with, 1033–1036
buffers
  circular, 1261
  file, 231
  keyboard, 91
`Button` control, 762, 763, 792–798
  and events handling, 794–798
byte, 4
byte code, 8
`byte` data type, 45 46
`Byte.parseByte()`, 96
`Byte` wrapper class, 96, 594

**C**

calling methods, 73, 272–276, 294–296
  layered method calls, 276–277
  superclass constructor, 624–630
call stack, 720, 723, 725
`Car` class, 387
Cascading Style Sheet (CSS)
  applying to JavaFX application, 826–830
  RGB color system, 832–834
  style properties, 825–826
  styling JavaFX applications with, 823–838
  type selectors, JavaFX nodes and corresponding, 824–825
case conversion, character, 558–565
case sensitive, 29, 43
`case` statements, 152–153
cast operators, 66–67
catch block, 704
`catch` clause, 704
  using multiple, 710–718
  using one, 721–723
CDs (compact discs), 5
`CellPhone` class example, 350–354
central processing unit (CPU), 7, 12
  fetch/decode/execute cycle, 3–4
  parts/organization of, 3
  role of, 3
chains of inheritance, 647–652
`Character` class
  case conversion, 563–565
  description of, 558–565
  `isDigit()`, 558
  `isLetter()`, 558
  `isLetterOrDigit()`, 558
  `isLowerCase()`, 558
  `isSpaceChar()`, 558
  `isUpperCase()`, 558
  `isWhiteSpace()`, 558
  `toLowerCase()`, 563
  `toUpperCase()`, 563
characters
  comparing, 119–120
  conversion, 163
  literals, 50–51
  reading, 89
  Unicode, 51–52, 119
`charAt()`, 74, 581
`char` data type, 50–52, 558
`CheckBox` controls, 848–853
  clicks, responding to, 853
  determination of selection, 848–849
  selection in code, 849–853
checked exceptions, 724–725
check menu item, 888
`CheckMenuItem` class, 888
Cho Han game example, 360–370
`Circle` class, constructors, 914–917
circles, drawing
  concentric, 1008–1010
circularly-linked list, 1223
`ClassCastException` class, 1081
classes. *See also specific classes*
  `accessor()`, 338
  access specifiers, 326
  aggregation, 515–525
  anonymous, 681–684
  base, 612
  body of, 29
  collaborations, 542–546
  constructors and, 346–350
  data hiding, 338–339
  definition, 28
  derived, 612

classes *(continued)*
  finding, 382–385
  generic, 1082–1090
  header, 28
  hierarchies of, 653
  inner, 681–684
  instance fields and, 341–345, 378–379, 493–494
  instance methods and, 328, 341–345, 493–494
  instance of, 341–345
  interfaces, 667–681
  in Java API, 319–321
  layout of members, 340
  mutator method, 338
  names, 29, 44
  objects created from, 318–319
  private access specifier, 326
  protected members, 641–646
  public access specifier, 326
  responsibilities, identifying, 385–388
  static, 493–499
  type variables, 71–72, 322
  writing, step-by-step instructions, 324–340
`.class` file extension, 28
`close()`, 734, 736
COBOL, 7
code reuse, 270
collaboration, class, 542–546
`Collection` interface, 1131–1133, 1172
  `addAll` method, 1131
  `add` method, 1131
  `clear` method, 1131
  `containsAll` method, 1131
  `contains` method, 1131
  `equals` method, 1131
  `removeAll` method, 1132
  `removeIf` method, 1132
  `remove` method, 1131
  `retainAll` method, 1132
  `size` method, 1132
  `stream` method, 1132
  `toArray` method, 1132
`Collections` class, 1169–1170
  static methods, 1170
`Collectors` class, 1185
collectors, stream API, 1184–1189
`ColorAdjust` class, 958, 962
colors
  in CSS, 832–834
  stroke, changing, 914
combined assignment operators, 63–64
`ComboBox` class
  `getValue()` method, 875–877
  list of methods, 875
`ComboBox` controls, 874–879
  editable, 879
  responding to item selection with event handler, 877–878
  retrieving selected item, 875–877
command-line arguments, 468–470
command line interface, 759
commas
  in numeric literals, 47, 48
  separated value, 597
  separators, 167–168
comments
  block, 80
  defined, 78
  documentation, 80–82, 277–278, 288–290, 296–297
  multi-line, 79–80
  single-line, 78–79
  slash marks, use of, 28, 31
compact discs (CDs), 5
`Comparable` interface, 1033, 1036, 1040, 1047, 1052, 1107–1110
  `compareTo` method, 1033, 1036
`Comparator` interface, 1331–1333
  abstract methods, 1156
  `compare` methods, 1156
  `equals` methods, 1156
`compareTo` method, 1033, 1036
comparing arrays, 426–428
comparing string objects, 142–147
compiler, role of, 12–13
compiling programs, 14–15
compound operators, 63
computer systems
  hardware, 2–5
  software, 5–6
`concat()`, 576
concatenation operator, 40–41
conditional expression, 149–150
conditional loops, 207
conditionally executed, 112, 118
conditional operator, 149–150
console
  interface, 759
  output, 33
  window, 33
constants
  `enum`, 531, 770
  `Interpolator`, 957
  `MAX_VALUE`, 598–599
  `MIN_VALUE`, 598–599
  named, 69–70
constructors
  `Arc` class, 924–928
  `Circle` class, 914–917
  copy, 514–515
  default, 348–349, 630–631
  `FadeTransition` class, 954
  `FillTransition` class, 953
  `Line` class, 911–914
  no-arg, 349, 630–631
  object creation and, 346–350
  overloading, 370–378, 530–531
  `Polyline` class, 931–933
  `Rectangle` class, 918–921
  `RotateTransition` class, 944
  `ScaleTransition` class, 949
  storage, 4–5
  `StringBuilder` class, 580–581
  `String` class, 350
  `StrokeTransition` class, 952
  superclass, 621–630
  `Text` class, 933
  `TranslateTransition` class, 941
  in UML diagrams, 348
consumer, 1126
contiguous allocation, 1195
`continue` statements, 229
contract, interface as, 669–673
control characters, 36
controls, JavaFX
  `Button` control, 762, 763, 792–798
  `CheckBox` controls, 848–853
  `ComboBox` controls, 874–879
  creating, 766
  `Label` control, 762, 763
  `ListView` control, 853–874
  `RadioButton` controls, 838–847
  `Slider` controls, 880–885
  `TextArea` controls, 885–887
  `TextField` control, 762, 763, 799–803
control unit, 3
control variable, loop, 196
conversion
  character, 163

    data type, 65–68, 96–99, 281–282
copy (copying)
    arrays, 420–422
    constructors, 514–515
    deep, 523–525
    devices for, 5
    objects, 512–514
    reference, 420–422
    shallow, 523–525
count-controlled loops, 207, 214–216
counter variables, 208
C programming language, 7
C# programming language, 7
C++ programming language, 7
CPU. *See* central processing unit
CRC cards, 545–546
`.csv` file, 597
currency symbols, 48
`Customer` class, 386
custom style classes, creating, 834–836

**D**

data
    appending, to files, 236–237, 738
    hiding, 20, 338–339
    reading, from files, 237–238, 736–738
    stale, 339
    writing, to files, 230–236, 733–735
`DataInputStream` class, 736–738
`DataOutputStream` class, 733–735
data to files, 230–236
data types
    `boolean`, 50
    `byte`, 45, 46
    `char`, 50–52
    conversion between, 65–68, 96–99, 281–282
    `double`, 45, 47
    enumerated, 531–540
    `float`, 45, 47
    `floating-point`, 47–49
    `int`, 45–47
    `long`, 45, 46
    mixing, in expressions, 67–68
    numeric data, 45, 597–599
    primitive, 44–54, 65–68
    ranking, 65–66
    `short`, 45, 46
    in UML diagrams, 339–340
decimal notation, 49
decision structures
    conditional operator, 149–150
    defined, 112
    `if-else-if` statement, 128–134
    `if-else` statement, 120–122
    `if` statement, 111–120
    logical operators, 134–141
    `nested if` statement, 122–128
    `switch` statement, 150–160
declarations
    array notation, 412
    parameters, lambda expressions, 687
    variable, 39, 147–148
decrement operator (++), 189–192
    postfix *versus* prefix modes, 192
deep copy, 523–524
`default()`, 674–676
default constructor, 348–349, 630–631
default error message, retrieving, 707–710
default exception handler, 702
default methods, interfaces, 674–676
`default` statements, 152–153
definition, class, 28
`delete()`, 583–584
`deleteCharAt()`, 583–584
delimiters, 232, 594
depth of recursion, 1001
derived class, 612
descending order, 1027
deserialization, 732, 745
dialog boxes, 93–99
diamond operator for type inference, 480, 1082
dice with objects, example, 354–358
digital versatile discs (DVDs), 5
digraphs, 1234–1238, 1279
directed graphs, 1234–1238, 1279
directory
    rooted tree, 1342
    searching, 1279–1281
direct recursion, 1006
disk drives, 4–5. *See also* solid state drives
`distinct()` operator, 1177
divide-and-conquer approach, 269
division
    integer, 57
    operator, 55, 56
    remainder of, 56
documentation comments, 80–82, 277–278, 288–290, 296–297, 731–732
dot, 34
`double` data type, 45, 47
`Double.parseDouble()`, 96
`Double` wrapper class, 96, 597
doubly-linked list, 1033, 1215–1223
`do-while` loops, 204–206
`drawCircles()`, 1009
drawing, concentric circles, 1008–1010
`DropShadow` class, 958–961
    methods, 960
DVDs (digital versatile discs), 5
dynamic binding, 656–658

**E**

edges, 1235
editor, text, 12
`Ellipse` class, constructors, 921–924
`else`, trailing, 131–132
encapsulation, 20
`EndOfMedia` event handlers, 971–974
`endsWith()`, 566–568
enhanced `for` loop, 416–418, 474–476, 1131
E notation, 49
enter key, 89
`enum` constants, 531, 770
enumerated data types, 531–539
    switching on, 538–539
`EOFException` class, 703
`equals()`, 142–144, 509–512, 654, 1156
erasure, 1113–1116
`Error` class, 724
error messages
    cannot resolve symbol, 77
    default error message, retrieving, 707–710
errors
    common, to avoid, 99–100, 255, 388–390, 481, 546, 689, 749, 1021–1022
    compiler, 292, 294

- errors *(continued)*
  - exception handler used for recovering from, 713–716
  - off-by-one, 410, 416
  - syntax, 12
  - using trailing `else` to catch, 131
- escape sequences, 36–37
- event-driven GUI programs, 761
- event handlers, 794
  - anonymous inner classes and, 803–805
  - `EndOfMedia`, 971–974
  - key events, 979–985
  - lambda expressions and, 806–808
  - mouse events, 986–991
  - registering, 795–798
  - responding to `ComboBox` item selection with, 877–878
  - responding to `ListView` item selection with, 859–860
  - writing, 794–795
- events
  - definition, 794
  - firing, 794
  - handling, with `Button` control, 794–798
  - object, 794
  - source, 794
- Excel spreadsheet, 597
- `Exception class`, description of, 702–703
- exception handler
  - defined, 702
  - recovering from errors using, 713–716
- exceptions
  - `catch` clauses, using multiple, 710–718
  - `catch` clause, using one, 721–723
  - checked, 724–725
  - `ClassCastException` class, 1081
  - classes, 702–703
  - creating your own classes for, 728–731
  - default error message, retrieving, 707–710
  - default handler, 702
  - defined, 233, 701, 702
  - `EOFException class`, 703
  - `Error` class, 724
  - `Exception` class, 702–703
  - `FileNotFoundException` class, 703
  - `finally` clause, 718–719
  - handling, 701–725
  - handling multiple, 710–718
  - `IllegalArgumentException` class, 718, 727–728
  - `IndexOutBoundsException`, 1135
  - `InputMismatchException` class, 722
  - `IOException` class, 233, 235, 237, 241, 703, 722
  - `NoSuchElementException`, 1129, 1131, 1137, 1141, 1174
  - `NullPointerException`, 1145, 1160, 1167
  - `NumberFormatException` class, 717–718, 721–722
  - polymorphism and, 710
  - `RuntimeException` class, 703, 724, 728
  - stack trace, 720–721
  - throwing, 725–732
  - `throws` clause, 233, 241
  - `try` statement, 703–707, 717–718
  - unchecked, 724–725
  - `UnsupportedOperationException`, 1159
  - when not caught, 723–724
- executable files, 12
- `exists()`, 245–248
- explicit `import` statement, 380
- expressions
  - initialization, 207–208, 211–214
  - regular, 591
  - `return` statement, 294
  - test, 207–208
  - update, 207–208, 211, 213–214
- `extends` key word, in generics, 1092–1095, 1099

**F**

- `factorial()`, 1003–1006
- `FadeTransition` class, 954–955
- fetch/decode/execute cycle, 3–4
- Fibonacci series, 1011–1012
- fields
  - instance, 341–345, 378–380, 493–495
  - interface, 673
  - `length`, 416, 444, 455–456
  - object, 317
  - static, 494–496
- `FileChooser` class, 897–898
  - `showOpenDialog` method, 898
  - `showSaveDialog` method, 898
- `File` class, 230
  - `exists()`, 245–248
- `FileInputStream` class, 736–737
- `FileNotFoundException` class, 703
- `FileOutputStream` class, 733–735, 738
- files
  - appending data to, 236–237, 738
  - arrays and, 438–439
  - binary, 230, 732–738
  - buffers and, 231
  - closing, 230
  - comma separated value format, 597
  - detecting end of, 241–242
  - exceptions, 233
  - existence, checking for, 245–248
  - `FileWriter` class, 230, 236–237
  - input/output, 230–248
  - location of, specifying, 237
  - names, 14
  - opening, 230
  - pointer, 741–744
  - `PrintWriter` class, 230–236
  - random access, 739–744
  - reading, 230
  - reading data from, 237–238, 736–738
  - reading lines from, using `nextLine()`, 238–241
  - reading primitive values from, 242–244
  - read position of, 239
  - `Scanner` class, 230, 237–244
  - sequential access to, 739
  - source, 12
  - text, 230
  - `throws` clause, 233, 241
  - writing data to, 230–236, 733–735
- `FileWriter` class, 230, 236–237
- `FillTransition` class, 953–954
- `filter()`operator, 1177

`finalize()`, 542
`final` key word, 69–70
finally block, 719
`finally` clause, 718–719
flags, 118, 167–170
flash memory, 5
`float` data type, 45, 47
floating-point data types, 47–49
floating-point literals, 48–49
`Float.parseFloat()`, 96
`Float` wrapper class, 96, 594
floppy disk drives, 5
`for` loops
  `ArrayList` class and, 475–476
  count-controlled, 207, 214–216
  counter variable, 208
  defined, 207
  enhanced, 416–418, 475–476, 1131
  header, 207
  initialization expression, 207–208, 211–214
  pretest, 210
  test expression, 207–208
  update expression, 207–208, 211, 213–214
  user-controlled, 212–213
formal parameters, 281
`FormatString`, 172
formatting
  string arguments, 170–171
  `String.format()`, 172–173
  `System.out.printf()`, 160–174
FORTRAN, 7
function, 1126
functional decomposition, 302
functional interfaces, 684–688, 1126. *See also* lambda expressions
functional RGB notation, 833

**G**
garbage collection, 540–542
  `finalize()`, 542
`GaussianBlur` class, 958, 964
generalization and specialization, 611–612
`generate()`, 1172
generics
  `ArrayList` class, 1079–1081
  `Comparable` interface, and, 1107–1110
  constraining a type parameter, 1092–1094, 1096–1100
  erasure, 1113–1116
  `extends` key word, 1092–1095, 1099
  generic class, writing, 1082–1090
  generic methods, writing, 1096–1097
  inheritance and, 1100–1103
  interfaces, and, 1107–1110, 1172
  introduction to, 1079–1081
  JFC, 1126
  lower bound constraining, 1095
  passing generic class objects to methods, 1090–1095
  raw type, 1088–1089
  restrictions on use of, 1116–1118
  subsets, 1111–1113
  `super` key word, 1095
  type parameter, 1084–1085, 1090, 1094–1095, 1104–1106
  type parameter, multiple, 1104–1106
  type wildcard, 1091–1092
  upper bound constraining, 1095
  ? wildcard, 1091
`get()`, 473–475, 1162
`getArea()`, 335–338
`getChars()`, 572–575, 581
`getKey()`, 1164
`getLength()`, 332–335
`getSelectionModel().getSelectedIndex()` method, 855–857, 863
`getSelectionModel().getSelectedItem()` method, 855–856, 863
getters, 338
`getValue()`, 1164
`getWidth()`, 332–335
`Glow` class, 958, 966
Gosling, James, 8
graphical user interface (GUI), 759–761
  application, debugging, 898–902
`Graphics` class, `drawCircles()`, 1009
greatest common divisor (GCD), finding, 1012–1013
Green Team, 8
`GridPane` layout container, 784–792
GUI. *See* graphical user interface
GUI (graphical user interface), 759–761
  application, debugging, 898–902

**H**
hard drives, 4–5
hardware
  central processing unit (CPU), 3–4, 7, 12
  components of, 2
  defined, 2
  input devices, 5
  memory, 4
  output devices, 5
  secondary storage, 4–5
hash code, 654, 1143, 1145, 1158
  collision, 1144
  `hashCode` method, 1144, 1148–1152
hashing algorithm, 1143–1144
`HashMap` class, 1159–1166
  constructors, 1160
  `entrySet` method, 1164–1166
  `get` method, 1162
`HashSet` class, 1144–1152
  constructors, 1145
  initial capacity, 1145
  iterator, 1147
  load factor, 1145
`HBox` layout container, 767, 777–782
headers
  class, 28
  loop, 194, 207
  method, 29–30, 233, 241, 271–272
  `while` loop, 194
heap, 1335
  adding an item to, 1336–1337
  removing the minimum element, 1337–1342
Heapsort algorithm, 1330–1333
hexadecimal numbers, 654–655
hiding, data, 20, 338–340
hierarchies, class, 653
HotJava, 8
HTML. *See* Hypertext Markup Language

Hypertext Markup Language (HTML), 80

**I**

identifiers, 9, 42–43
IDEs (integrated development environments), 14
ID selector, 836
`if-else-if` statements
  compared to nested decision structure, 132–133
  description of, 128–134
  trailing `else` clause, 131–132
`if-else` statement, 120–122
`if` statement
  braces used with, 118
  characters, comparing, 119–120
  description of, 111–120
  flags, 118
  flowcharting, 112–113
  multiple statements, 118
  nested, 122–128
  programming style and, 117
  relational operators to form conditions, 113–114
`IllegalArgumentException` class, 718, 727–728
`Image` class, 772–776
  `setPreserveRatio` method, 775–776
images
  aspect ratio, preserving, 775–776
  displaying, 772–776
  loading from Internet, 775
  size of, 775
`ImageView` class, 773–776
  `setImage` method, 776
immutable objects, 286, 524
implementing an interface, 667–673, 684–688
`import` statement
  explicit, 380
  `Scanner` class and, 88–89
increment operator (`++`), 189–192
  postfix *versus* prefix modes, 192
indentation, 82
`indexOf()`, 569, 581
indirect recursion, 1006
infinite loops, 196–197
inheritance
  base class, 612
  chains of, 647–652
  defined, 611
  derived class, 612
  does not work in reverse, 623–624
  fields, 493–495
  functional, 684–688
  generalization and specialization, 611–612
  generics and, 1100–1103
  implementing, 684–688
  "is a" relationship, 612–620, 658–659
  methods, 493–495
  subclasses and, 612
  superclass, 612
  superclass constructors, 621–630
  superclass methods, overriding, 632–637
  UML diagrams, 620–621
initialization
  array, 411–412
  array, two-dimensional, 454–455
  variable, 52–53
inline style rules, 837
inner class
  anonymous, 681–684
`InnerShadow` class, 958, 961
input
  devices, 5
  dialogs, displaying, 94
  file, 230–248
  keyboard, 85–92
  validation, 200–203
`InputMismatchException` class, 711, 722
`insert()`, 582–583
inserting, items in `ArrayList`, 473, 477–478
insertion sort, 1027, 1042–1047
  Big O notation, 1070
  pseudocode, 1045
  sorting objects with, 1047
instance
  class, 71, 341–345
  fields, 341–345, 378–379
  methods, 328, 341–345
  variables, 345
`instanceof` operator, 659
`InStream` interface, 1173
`InStream` objects, 1173
`int` data type, 39, 45–47
`Integer.parseInt()`, 96
integers
  data types, 45–47
  division, 57
  literals, 42, 47
  mixed, 67–68
`Integer` wrapper class, 96, 597
integrated development environments (IDEs), 14
interfaces. *See also* graphical user interface
  as contract, 669–673
  default methods, 674–676
  defined, 667
  fields, 673
  generics and, 1107–1110
  implementing, 667–673
  key word, 667
  multiple, 673
  `ObservableList`, 812–813
  polymorphism and, 676–681
  serializable, 748
  UML diagram, 673–674
Internet Explorer, 80
`IOException` class, 233, 235, 237, 241, 703, 722, 734, 745
"is a" relationship, 612–620, 658–659
`isDigit()`, 558
`isLetter()`, 558
`isLetterOrDigit()`, 558
`isLowerCase()`, 558
`isSelected` method, 839
`isSpaceChar()`, 558
`isUpperCase()`, 558
`isWhiteSpace()`, 558
`Iterable` interface, 1127–1128
  `forEachRemaining` method, 1129
  `hasNext` method, 1129
  `next` method, 1129
  `remove` method, 1129
iterations
  JFC, 1127
  loop, 195–196
iterators, 1128–1131

**J**

Java
  compiler, 12–13
  defined, 7
  editions, 13
  history of, 8
  parts of a simple program, 27–31

- portability, 12
- virtual machine, 12–13, 540, 657, 703, 704

- `java.applet`, 381
- `java.awt`, 381
- Java Collections Framework (JFC), 1125–1133
  - `Collections` class, 1169–1170
  - `Collectors` class, 1185
  - enhanced `for` loop, 1131
  - interfaces, 1126–1127
  - `Iterable` interface, 1127–1128
  - iterators, 1128–1131
  - lists, 1125–1126, 1133–1142
  - maps, 1126, 1158–1169
  - sets, 1126, 1143–1158
  - stream, 1171–1189
- `java` command, 14, 28, 469
- Java Database Connectivity. *See* JDBC
- Java Development Kit (JDK), 13
- `javadoc`, documentation and, 80–82
- Java Enterprise Edition (EE), 13
- `java` file extension, 28
- `.java` file extension, 12
- JavaFX, 762
  - `Application` class, 763–765
  - applications styling with CSS, 823–838
  - `BoxBlur` class, 958, 963
  - `ColorAdjust` class, 958, 962
  - custom style classes, creating, 834–836
  - `DropShadow` class, 958–961
  - `FileChooser` class, 897–898
  - `GaussianBlur` class, 958, 964
  - `Glow` class, 958, 966
  - `Image` class, 772–776
  - images displaying, 772–776
  - `ImageView` class, 773–776
  - `InnerShadow` class, 958, 961
  - `Line` class, 911–914
  - `Media` class, 969–974
  - `MediaPlayer` class, 969–974
  - `MediaView` class, 974–979
  - menus, 887–895
  - `MotionBlur` class, 958, 964–965
  - nodes, and corresponding CSS type selectors, 824–825
  - `ObservableList` interface, 812–813
  - `Reflection` class, 958, 967
  - `SepiaTone` class, 965–966
  - `Shape` class, 910–911
  - stages, 763, 765, 768–770
- `java.io`, 381, 703, 733, 736, 745
- `java.lang`, 381
- Java Micro Edition (ME), 13
- `java.net`, 381
- JavaScript, 7
- `java.security`, 381
- `java.sql`, 381
- Java Standard Edition, (SE), 13
- `java.text`, 381
- `java.util`, 381, 590
- `java.util.Random` class, 1173
- `java.util.stream` package, 1172
- Java Virtual Machine (JVM), 12–13, 540, 657, 703, 704
- `javax.swing`, 381
- JDK (Java Development Kit), 13
- `JOptionPane` class
  - dialog boxes displayed using, 93–99
  - `showInputDialog()`, 94
  - `showMessageDialog()`, 93–94
- `JTree` component, 1342–1345
- JVM. *See* Java Virtual Machine

**K**

- keyboard
  - buffer, 91
  - input from, 85–92
- `KeyEvent` class, 979–980
- `KEY_PRESSED` event handler, 980–985
- `KEY_RELEASED`, 979–985
- `keySet` method, 1163
- `KEY_TYPED`, 979–985
- key words, 9, 10, 42
  - super, 624–630, 640

**L**

- `Label` control, 762, 763
- lambda expressions, 684–688
  - accessing variables within, 688
  - declaration, parameter, 687
  - and event handlers, 806–808
  - multiple statements in, 687
  - parameters, multiple, 687
- languages, programming
  - common elements, 8–15
  - description of, 6–8
- `lastIndexOf()`, 569–575, 581
- late binding, 657
- layered method calls, 276–277
- layout containers, 766–767, 776
  - `BorderPane`, 808–812
  - `GridPane`, 784–792
  - `HBox`, 767, 777–782
  - multiple, in same screen, 791–792
  - `VBox`, 782–784
- leading whitespace, 577
- leading zeros, 168–169
- left-justifying numbers, 170
- `length()`, 73, 74, 581
- `length` field, array, 416, 444, 455–457
- lexicographical comparison, 145
- lifetime, 292
- `limit()`, 1177
- `Line` class, 911–914
  - constructors, 911
- lines
  - blank, 28
  - program, 11
  - reading, from files, using `nextLine()`, 238–241
- linked allocation, 1195
- `LinkedHashMap` class, 1159, 1166–1168
  - constructors, 1167
- `LinkedHashSet` class, 1152–1153
- linked list, 1195–1238
  - adding nodes to, 1196–1198, 1216–1217
  - `add` method, 1203, 1225–1226
  - circularly-linked, 1223
  - creating, 1196–1198
  - doubly-linked list, 1215–1223
  - empty list, 1196
  - graphical interfaces, 1210–1214
  - head, 1196
  - head reference, 1196
  - `isEmpty` method, 1202–1203
  - node, 1195–1196
  - operations, 1202–1210
  - queue, implementation of, 1270–1275
  - recursive algorithms for, 1224–1238
  - `remove` method, 1202, 1204, 1226–1234
  - removing nodes, 1198–1199, 1217–1223

- linked list *(continued)*
  - `size` method, 1203, 1224–1225
  - tail, 1224
  - `toString` method, 1204
  - traversing, 1199–1201
- `LinkedList` class, 1135–1137, 1140–1141
  - `addFirst` method, 1141
  - `addLast` method, 1141
  - `getFirst` method, 1141
  - `getLast` method, 1141
  - `removeFirst` method, 1141
  - `removeLast` method, 1141
- Linux, 5
- `List` interface, 1134–1135
  - `addAll` method, 1134, 1135
  - `add` method, 1134, 1135
  - `get` method, 1134, 1135
  - `indexOf` method, 1134
  - `lastIndexOf` method, 1134
  - `listIterator` method, 1137–1140
  - `remove` method, 1134, 1135
  - `set` method, 1134, 1135
  - `sort` method, 1134
  - `subList` method, 1134
- lists
  - adjacency, 1235–1237
- lists, JFC, 1125–1126, 1133–1142
  - creating and initializing, 1136–1137
  - iterators, 1137–1140
- `ListView` class
  - `getItems().addAll()` method, 860–861
  - `getItems().setAll()` method, 860–861
  - `getSelectionModel().getSelectedIndex()` method, 855–857, 863
  - `getSelectionModel().getSelectedItem()` method, 855–856, 863
  - methods to select items, 868
  - retrieving selected item, 855–856
  - selected item, retrieving index of, 855–857
- `ListView` control, 853–874
  - adding items *vs.* setting items, 860–861
  - change event, 859
  - initializing with array/`ArrayList`, 861–862
  - methods to select items, 868
  - orientation, 868
  - preferred size of, 854
  - retrieving multiple selected items, 863–866
  - selection modes, 862–863
- literals
  - character, 50–51
  - floating-point, 48–49
  - how to use, 39–44
  - integer, 42, 47
  - string, 30, 42
- load factor, `HashSet` class, 1145, 1160, 1167
- local variables, 76, 290–292
- logical, 17
- logical errors, 17
- logical operators
  - `&&` (AND), 134–137
  - associativity of, 139–140
  - `boolean` expressions using, 135
  - `!` (NOT), 134, 139
  - numeric ranges with, 140–141
  - `||` (OR), 134, 137–139
  - precedence of, 139–140
- `long count()`, 1173
- `long` data type, 45, 46
- `Long.parseLong()`, 96
- `LongStream interface`, 1173
- `Long` wrapper class, 96, 594
- loops
  - body, 194
  - `break` statement in, 229
  - choice of, 229
  - conditional, 207
  - `continue` statement in, 229
  - control variable, 196
  - count-controlled, 207, 214–216
  - defined, 193
  - `do-while`, 204–206
  - enhanced, 416–418, 474–476
  - enhanced `for`, 416–418, 474–475
  - `for`, 207–216
  - header, 194, 207
  - infinite, 196–197
  - iterations, 195–196
  - nested, 221–229
  - posttest, 204
  - pretest, 196, 210
  - running totals and, 216–219
  - sentinel values and, 219–221
  - user-controlled, 206, 212–213
  - `while`, 193–206
- lower bounds, generics, 1095
- lowercase, converting characters to, 558

**M**

- machine language, 6–7
- Mac OS, 5
- `main()`
  - body of, 30
  - header, 30
- `Map.Entry` interface, 1159, 1164
- `Map` interface, 1159–1160
  - `clear` method, 1159
  - `containsKey` method, 1159
  - `containsValue` method, 1159
  - `entrySet` method, 1159, 1164–1166
  - `get` method, 1159, 1162
  - `isEmpty` method, 1159
  - `keySet` method, 1159, 1163
  - `put` method, 1160
  - `remove` method, 1160
  - `size` method, 1160
  - `values` method, 1160
- `map` operation, stream API, 1181
- `map()`operator, 1177
- `mappings`, 1159
- `maps`, 1126, 1158–1169
- `Math` class
  - `.PI` named constant, 70
  - `.pow`, 62
  - `.sqrt`, 62
- `MAX_VALUE` constant, 595
- `Media` class, 969–974
- `MediaPlayer` class, 969–974
  - methods, 970
- `MediaView` class, 974–979
- memory
  - flash, 5
  - random-access, 4, 11
  - secondary, 4–5
  - storing characters in, 52
- menu bar, 888
- `MenuBar` class, 888
- `Menu` class, 888
- menu item, 888
- `MenuItem` class, 888
- menus, 887–896
  - components of, 888
  - JavaFX, 887–896
- `message()`, 1001
- message dialogs, displaying, 93–94

methods. *See also specific methods*
  abstract, 660–667
  accessor, 338
  arguments, passing, 279–289, 500–503
  binding, 371
  body, 271
  calling, 73, 272–276, 294–296
  defined, 20, 269
  defining a `void`, 271–272
  detail section, 81
  divide-and-conquer approach, 269
  documentation comments, 277–278, 288–290, 296–297
  generic, 1096–1097
  headers, 29–30, 233, 241, 271–272
  hiding, data, 338–340
  how to use, 269–270, 297–300
  instance, 328, 341–345, 493–495
  `javadoc` comments with, 278
  layered method calls, 276
  local variables, 76, 290–292
  modifiers, 272
  modifiers, method, 272
  mutator, 338
  name, 272
  object references/arguments to methods, passing, 285–288, 500–503
  overloading, 370–378
  overriding, 632–637
  parameter variables and, 281–282
  passing arrays to, 422–424, 459–461
  private, 326
  problem solving with, 302–305
  public, 272, 326
  returning a `boolean` value, 300
  returning arrays from, 439–441
  returning objects from, 503–505
  `return` statement, 294
  return type and, 293
  static, 272, 497–499
  summary section, 81
  synchronized, 589
  value-returning, 269–271, 293–297
  void, 269–272
Microsoft, 7
minimum field width, 164–166
minus sign (negation), 55
`MIN_VALUE` constant, 595
mnemonics, 895–896
modulus operator, 55, 56
`MotionBlur` class, 958, 964–965
`MouseEvent` class, 986–991
mouse events, 986–991
multi-catch, 722–723
multi-line comments, 79–80
multiple statements in, 687
multiplication operator, 55
multithreaded application, 589
mutator method, 338

**N**

named colors, 833–834
named constants, 69–70
names
  class, 29, 44
  identifiers, 42–43
  method, 272
  variable, 43
narrowing conversion, 66
negation operator, 55
nested `if` statements
  description of, 122–128
  flowcharting, 122–123
  multiple, 125–128
nested loops, 221–229
`new` key word, 85, 403, 406, 411, 442, 454, 505, 581
`nextByte()`, 87, 242
`nextDouble()`, 87, 242, 250
`nextFloat()`, 87, 242, 250
`nextInt()`, 87, 242, 250
`nextInt(int n)`, 250
`nextLine()`, 87, 238–242
`nextLong()`, 87, 242, 250
`nextShort()`, 87, 242
no-arg constructor, 349, 630–631
`Node` class, 936–938
nodes, JavaFX
  `BoxBlur` class, 958, 963
  `ColorAdjust` class, 958, 962
  combining effects, 967–969
  CSS type selectors, 824–825
  `DropShadow` class, 958–961
  `GaussianBlur` class, 958, 964
  `Glow` class, 958, 966
  ID, 836
  `InnerShadow` class, 958, 961
  `Line` class, 911–914
  `MotionBlur` class, 958, 964–965
  `Reflection` class, 958, 967
  root, applying styles to, 830–831
  rotating, 936–938
  scaling, 938–939
  `SepiaTone` class, 958, 965–966
  `Shape` class, 910–911
  style properties, 825–826
NOT (`!`) operator, 134, 139
null references, avoiding, 525–527
`null` statements, 117
`NumberFormatException` class, 710, 717–718, 721–722
numbers, converting strings to, 96–99
numeric data, data types for, 45
numeric data types, wrapper classes for, 594–596
numeric ranges, checking with logical operators, 140–141

**O**

Oak, 8
`Object` class
  description of, 653–655
  equals(), 654
  toString(), 654
`ObjectInputStream` class, 745
object-oriented design
  class collaboration, 542–544
  classes, finding, 382–385
  class responsibilities, identifying, 385–388
  CRC cards, 545–547
  problem domain, 382
object-oriented programming (OOP)
  attributes in, 20
  data hiding in, 20, 338–339
  defined, 19–21
  encapsulation in, 20
  methods in, 20
`ObjectOutputStream` class, 745
objects
  aggregation, 515–517
  as arguments/references, passing, 285–288, 358–360, 500–502
  `ArrayList`, creating and using, 473–475

objects *(continued)*
  arrays of, 433–435, 444–446
  comparing string, 142–147
  constructors, 346–350, 514–516
  copy constructors, 514–516
  copying, 512–514
  created from classes, 71
  creating string, 72–73
  default constructor, 348–349
  defined, 317
  deserialization, 732, 745
  examples of, 318
  fields, 317
  immutable, 286, 524
  `out`, 33–34
  primitive variables *versus*, 321–324
  returning, from a method, 503–505
  returning reference to, 300–301
  serialization, 744–748
  sorting arrays o, 1033–1036, 1040–1042, 1047
  state, 505
`ObservableList`
  converting to array, 867
  elements of, 866–867
`ObservableList` interface, 812–813
`of()`, 1172
off-by-one errors, 410, 416
one-dimensional arrays, 450
OOP. *See* object-oriented programming
operands, 52
operating systems, 5
operation, stream API
  intermediate, 1172, 1177–1181
  pipeline of, 1171–1172
  terminal, 1172–1177
operators, 1126
  `=` (assignment), 52
  `+=` (assignment), 64
  `-=` (assignment), 64
  `*=` (assignment), 64
  `/=` (assignment), 64
  `%=` (assignment), 64
  binary, 55
  cast, 66–67
  combined/compound assignment, 63–64
  conditional, 149–150
  `-`(decrement), 189–192
  defined, 9, 10
  `<>`(diamond), 480
  `++` (increment), 189–192
  instance of, 659
  ternary, 55, 149
  unary, 55
operators, arithmetic
  `+` (addition), 55
  associativity of, 58
  `/` (division), 55, 56
  `%` (modulus), 55, 56
  `*` (multiplication), 55
  `-`(negation), 55
  precedence of operations, 57–59
  `+` (string concatenation), 40–41
  `-`(subtraction), 55
operators, logical
  `&&` (AND), 134–137
  associativity of, 139–140
  `boolean` expressions using, 135
  `!` (NOT), 134, 139
  numeric ranges with, 140–141
  `||` (OR), 134, 137–139
  precedence of, 139–140
operators, relational
  `==` (equal to), 113–114
  `>` (greater than), 113–114
  `>=` (greater than or equal to), 113–114
  `<` (less than), 113–114
  `<=` (less than or equal to), 113–114
  `!=` (not equal to), 113–114
optical devices, 5
Oracle/Sun Microsystems, 13
ordinal value, 533
OR (`||`) operator, 134, 137–139
`out` object, 33–34
output
  devices, 5, 33
  file, 230–248
overloading
  methods and constructors, 370–378, 530
  overriding *versus*, 637–640
overriding
  overloading *versus*, 637–640
  preventing, 640
  superclass methods, 632–637

**P**
package access, 646–647
packages
  `import` statement and, 380–381
  stream, 1172
padding, `HBox` layout container, 780–782
parameter variables (parameters), 279
  `@param`, 288–289
  data type compatibility, 281–22
  initializing local variables with, 292
  lambda expressions, 688
  list, 282
  other names for, 281
  scope, 282
  type, in generics, 1084–1085, 1090, 1094–1095, 1104–1106
  in UML diagrams, 339–341
  vararg, 470–472
parentheses, 31, 59, 149, 272
parse methods, 96, 557, 594
parsing, 1237
partially filled arrays, 437–439
Pascal, 7
passed by value, 284–285
`peek()`, 1178–1181
percentages, calculating, 59–62
Perl, 7
PHP, 7
`PI` constant, 70
`Polygon` class, 928–931
`Polyline` class, 931–933
polymorphism
  defined, 656
  description of, 655–660
  dynamic binding, 656–658
  exceptions and, 710
  interfaces, 676–681
portability, 12
postfix mode, 190–192
posttest loops, 204
precedence, operator, 57–59, 139–140

precision, 164
predecessor, 1195, 1287
predicate, 1126
preferred size, scene, 770
prefix mode, 190–192
pretest loops, 196, 210
priming read, 201, 221
primitive data types, 44–54
  conversion between, 65–68
  *versus* objects, 321–324
primitive type variables, 71–72
primitive values from files, reading, 242–244
primitive variables *versus* objects, 321–324
`print()`, 35–37
  patterns with nested loops, 224–229
  `PrintWriter` class and, 231, 232
`printf()`, 160–171
`println()`, 33–37
`PrintWriter` class
  `print()`, 231, 232
  `println()`, 231, 232
  writing data to files using, 230–236
`PriorityQueue` class, 1329
  `add` method, 1329
  comparators with, 1331–1333
  constructors, 1329
  `peek` method, 1329
  `poll` method, 1329
  `size` method, 1329
priority queues, 1328–1342
  `add` method, 1329
  application, 1330
  head of the queue, 1329
  Heapsort algorithm, 1330–1333
  implementation of, 1333
  `removeMin` method, 1329
`private` access specifier, 326
problem domain, 382
problem solving
  methods and, 302–305
  recursion and, 1002–1005
procedural programming, 19–20
procedure, 19
programmer-defined names, 9, 10
programmers, role of, 2
programming
  common elements, 8–15
  defensive, 539
  languages, 6–8
  object-oriented, 19–21
  process, 16–19
  style, 83–85, 117, 198
programs
  compiling running, 14–15
  defined, 6–7
  design with count-controlled loops, 207, 214–216
  design with while loop, 198–199
  parts of simple, 27–31
protected access specifier, 641–646
protected members, 641–646
pseudocode, 19
`public` access specifier, 28, 29, 326
`public` modifier, 272
punctuation, 9, 11
Python, 7

**Q**

queue data structures, 1259–1275
  array implementation of, 1259–1266
  circular buffer, 1261
  `dequeue` method, 1259
  `empty` method, 1259
  `enqueue` method, 1259
  front, 1259
  GUI demonstration of, 1266–1270
  last-in-first-out, 1259
  linked list implementation of, 1270–1275
  `peek` method, 1259
  priority (*see* priority queues)
  queue invariant, 1259
  rear, 1259
Quicksort, 1027, 1048–1052
  Big O notation, 1069–1070
  pivot, 1048
  pseudocode, 1049
  sorting objects with, 1052
quotation marks, 31, 37, 40, 41

**R**

`RadioButton` class
  `isSelected` method, 839
  `setSelected` method, 839–844
`RadioButton` controls, 838–847
  action event, responding to, 844–847
  determination of selection, 839
  selection in code, 839–844
radio menu item, 888
`RadioMenuItem` class, 888
ragged arrays, 461
RAM (random-access memory), 4, 11
`RandomAccessFile` class
  constructor, 739
  file pointer, 741–744
  format, 739
  reading and writing with, 740–741
  `seek()`, 742
random access files, 739–744
random-access memory (RAM), 4, 11
`Random` class, 1172
  how to use, 249–254
  `nextDouble()`, 250
  `nextFloat()`, 250
  `nextInt()`, 250
  `nextInt(Int n)`, 250
  `nextLong()`, 250
random numbers
  applications, 249
  generating, 249–254
`rangeClosed()`, 1173
range, numeric, 140–141
`rangeSum()`, 1007–1008
raw binary format, 733
`readBoolean()`, 736
`readByte()`, 736
`readChar()`, 736
`readDouble()`, 736
`readFloat()`, 736
reading
  data from files, 237–238, 736–738
  lines from files using `nextLine()`, 238–241
  primitive values from files, 242–244
  `RandomAccessFile` class and, 740–741
  strings, 738
`reading()`, 736
`readLong()`, 736
`readObject()`, 745
read position, 239
`readShort()`, 736

`readUTF()`, 736
`Rectangle` class, constructors, 918–921
`Rectangle` class, writing example, 324–340
recursion
  base case and, 1006
  `binarySearch()`, 1014–1017
  defined, 999–1002
  depth, 1001
  direct, 1006
  `factorial()`, 1003–1006
  Fibonacci series, 1011–1012
  greatest common divisor (GCD), finding, 1012–1013
  indirect, 1006
  linked lists, with, 1224–1238
  methods, 999–1002
  problem solving with, 1002–1005
  recursive case, 1007
  summing array elements with, 1007–1008
`recursiveBinarySearch()`, 1014–1017
recursive case, 1007
`reduce` methods, stream API, 1182–1184
reference copy, 420, 512
reference variables, 72
  objects and, 320–323
  reassigning array, 419–421
  `this`, 528–530
  uninitialized, 348
`Reflection` class, 958, 967
`regionMatches()`, 566, 568–569
regular expression, 591, 592
relational operators
  == (equal to), 113–114
  > (greater than), 113–114
  >= (greater than or equal to), 113–114
  < (less than), 113–114
  <= (less than or equal to), 113–114
  != (not equal to), 113–114
relationships
  “is a, 612–620, 658–659
  whole-part, 519
removing items from `ArrayList`, 476–478
`replace()`, 576–577, 583
replacing items in `ArrayList`, 478–480
reserved words, 9, 10
responsibilities, identifying class, 385–388
return
  arrays from methods, 439–441
  `@return`, 296–299
  `boolean` value, 300
  objects from methods, 503–505
  statements, 294
  type, 293–294
RGB color system, 832–834
rooted trees, 1342–1345
root selector, 830–831
`RotateTransition` class, 944–948
  constructors, 944
  `setFromAngle` method, 944–945
  `setToAngle` method, 944–945
Ruby, 7
running programs, 14–15
running totals, 216–219
`RuntimeException` class, 724, 728

**S**

`ScaleTransition` class, 949–951
  constructors, 949
`Scanner` class
  characters, reading, 89
  `close()`, 238
  `import` statement, 88–89
  mixing calls, 89–92
  `nextByte()`, 87, 242
  `nextDouble()`, 87, 242
  `nextFloat()`, 87, 242
  `nextInt()`, 87, 242
  `nextLine()`, 87, 238–241
  `nextLong()`, 87, 242
  `nextShort()`, 87, 242
  reading files with, 230, 237–244
  reading keyboard input, 85–92
`Scene` class, creating object of, 767
scene graph, 763, 765
scenes, JavaFX, 763
  creating, 765–772
  size of, 770
scientific notation, 49
scope
  coming into, 149
  instance fields, 378–380
  leaving, 149
  parameter variable, 282
  variable, 76–78, 147–148
screen coordinate system, 909–910
SDK (Software Development Kit), 13
search algorithm, 1052–1062
  binary search, 466–468, 1055–1063
  binary search, recursive, 1014–1017
  sequential search, 447–449, 1055–1063
secondary memory, 4–5
security
  aggregate classes and, 523–525
`seek()`, 742
selection sort, 1027, 1036–1042
selection sort algorithm, 463–465, 1027, 1036–1042
  Big O notation, 1070
  pseudocode, 1038
  sorting objects with, 1040–1042
  worst case complexity, 1066–1067
self-documenting program, 43
semicolons, 30–31, 117–118, 272
sentinel values, 219–221
separator bar, 888
`SepiaTone` class, 958, 965–966
sequence structure, 112
sequential file access, 739
sequential search algorithm, 447–449, 1052–1055, 1063
  average case analysis of, 1055
  Big O notation, 1069
  complexity of, 1065
  pseudocode, 1053
`Serializable` interface, 748
serialization, 744–748
`ServiceQuote class`, 387–389
`set()`, `ArrayList`, 478–480
`setByX` method, 942
`setByY` method, 942
`setCharAt()`, 583–584
`setFromAngle` method, 944–945
`setFromX` method, 941
`setFromY` method, 941

- `setHgap` method, 788
- `setId` method, 836
- `setImage` method, 776
- `Set` interface, 1143
- `setInterpolator` method, 956–957
- `setLength()`, 326–330
- `setOnMouseClicked` method, 986
- `setOnMouseDragged` method, 986
- `setOnMouseEntered` method, 986
- `setOnMouseExited` method, 986
- `setOnMouseMoved` method, 987
- `setOnMousePressed` method, 987
- `setOnMouseReleased` method, 987
- `setPadding` method
  - `HBox` class, 780
  - `VBox` class, 783
- `setPrefSize` method, 854
- `setPreserveRatio` method, 775–776
- `setRotate` method, 936–938
- sets, 1126, 1143–1158
- `setSelected` method, 839–844
- `setStroke` method, 914
- `setStyle` method, 837
- setters, 338, 340
- `setToAngle` method, 944–945
- `setToX` method, 941–942
- `setToY` method, 941–942
- `setWidth()`, 331–332
- shadowing, 379, 529–531
- shallow copy, 523–525
- `Shape` class, 910–911
  - `setStroke` method, 914
- shapes, drawing, 909–939
  - `Arc` class, 924–928
  - `Circle` class, 914–917
  - `Ellipse` class, 921–924
  - `Polygon` class, 928–931
  - `Polyline` class, 931–933
  - `Rectangle` class, 918–921
  - screen coordinate system, 909–910
  - stroke color, changing, 914
  - `Text` class, 933–935
- short-circuit evaluation, 135–136
- `short` data type, 45, 46
- `Short.parseShort()`, 96
- `Short` wrapper class, 96, 594
- `showInputDialog()`, 94
- `showMessageDialog()`, 93–94
- `showOpenDialog` method, 898
- `showSaveDialog` method, 898
- signature, 371
- simple Java program, 27–31
- single-line comments, 78–79
- size, array
  - declarator, 404–406
  - user specified, 417–419
- `skip()`operator, 1178
- slash marks, 28, 31
- `Slider` class, methods of, 881
- `Slider` controls, 880–885
- software
  - application, 5
  - categories, 5
  - defined, 2
  - engineering, 18–19
- Software Development Kit (SDK), 13
- solid state drives, 4, 5
- `SortedMap` interface, 1159, 1168–1169
- `sorted()`operator, 1178
- `SortedSet` interface, 1153–1155
- sorting algorithms, 1027–1052
  - ascending order, 1027
  - bubble sort, 1027–1036
  - descending order, 1027
  - Heapsort, 1330–1333
  - insertion sort, 1027, 1042–1047
  - introduction to, 1027
  - Quicksort, 1027, 1048–1052
  - selection sort, 463–465
- source code, 12
- source file, 12
- spaces, displaying, 36
- specialization, 611–612
- `split()`, 589–593
- spreadsheets, 5
- `Stack` class, 1246–1248
  - constructor, 1246
  - `empty` method, 1246
  - `peek` method, 1246
  - `pop` method, 1246
  - `push` method, 1246
- stack data structures, 1245–1258
  - array implementation, 1248–1253
  - `empty` method, 1246, 1249, 1255
  - generic implementation of, 1275–1278
  - last-in-first-out, 1245, 1248
  - objects, of, 1253
  - operations, 1246
  - `peek` method, 1246, 1249–1250, 1255
  - `pop` method, 1246, 1249–1250, 1255
  - `push` method, 1246, 1249
  - stack pointer, 1248
  - stack top pointer, 1248
  - top, 1245
- stack trace, 720–721
- stages, JavaFX, 763, 765, 768–770
- stale data, avoiding, 339
- standard input device, 85
- standard output device, 33
- star seven device (*7), 8
- `startsWith()`, 566–568
- statements, 11
  - assignment, 39
- state, object, 505
- static class members, 493–495
- static fields, 494–496
- static methods, 272, 497–499
- `static` modifier, 272
- `Stock` class, 506–508
- storage devices, types of, 4–5
- stream API
  - collectors, 1184–1189
  - creation, 1172–1173
  - defined, 1171
  - `map` operation, 1181
  - `reduce` methods, 1182–1184
  - terminal operation, 1172–1177
- `Stream` interfaces, 1172–1173
  - `distinct()` operator, 1177
  - `filter()` operator, 1177
  - `generate()`, 1172–1173
  - intermediate operators, 1177–1179
  - `limit()`operator, 1177
  - `map()`operator, 1177
  - `of()`, 1172
  - `peek()`, 1178–1181
  - `skip()`operator, 1178
  - `sorted()`operator, 1178
- stream reduction operators

- stream reduction operators *(continued)*
  - `boolean allMatch()`, 1174
  - `boolean anyMatch()`, 1174
  - `boolean noneMatch()`, 1174
  - `forEach()`, 1174
  - `optionalT max()`, 1174
  - `optionalT min()`, 1174
  - `toArray()`, 1174
- `String`, arguments, formatting, 170–171
- `StringBuffer` class, 589
- `StringBuilder` class
  - `append()`, 581–582
  - `charAt()`, 581
  - constructors, 580–581
  - defined, 579
  - `delete()`, 583–584
  - `deleteCharAt()`, 583–584
  - `getChars()`, 581
  - `indexOf()`, 581
  - `insert()`, 582–583
  - `lastIndexOf()`, 581
  - `length()`, 581
  - `replace()`, 583
  - `setCharAt()`, 583–584
  - `substring()`, 581
  - `toString()`, 584
- `String` class
  - arguments, formatting, 170–171
  - arrays of, 441–443
  - `charAt()`, 74
  - `compareTo()`, 144–146
  - `compareToIgnoreCase()`, 146
  - `concat()`, 576
  - constructor, 350
  - defined, 70–71
  - `endsWith()`, 566–568
  - `equals()`, 142–144
  - `getChars()`, 572–575
  - how to use, 70–76
  - `indexOf()`, 569
  - `lastIndexOf()`, 569, 573–575
  - `length()`, 73, 74
  - objects, creating, 72–73
  - objects of, passed as argument, 285–288
  - reading, 738
  - `regionMatches()`, 566, 568–569
  - `replace()`, 576–577
  - returning a `String object` from a method, 503–505
  - `split()`, 589–593
  - `startsWith()`, 566–568
  - `substring()`, 573
  - substrings, extracting, 572–575
  - substrings, searching for, 566–572
  - `toCharArray()`, 573–575
  - `toLowerCase()`, 75
  - `toUpperCase()`, 75
  - `trim()`, 576, 577
  - `valueOf()`, 577–578
  - writing, 738
- `String.format()`, 160, 172–173
- strings
  - concatenation operator, 40–41
  - converting, to numbers, 96–99
  - defined, 70
  - literals, 30, 42
  - tokenizing, 589–593
- `StrokeTransition` class, 952–953
- strongly typed language, 48, 65
- style, programming, 83–85, 117, 198
- style rules, 824
  - inline, 837
- style sheet, 823
- subclass, 612
- submenu, 888
- subscripts, 405, 449
- subsets, generic, 1111–1113
- `substring()`, 573, 581
- substrings
  - extracting, 572–575
  - searching for, 566–572
- subtraction operator, 55
- successor, 1195
- Sun Microsystems, 7, 8
- SunWorld, 8
- superclass, 612
  - constructors, 621–630
  - methods, overriding, 632–637
- `super` key word, 624–630, 640
  - in generics, 1095
- supplier, 1126
- switches, 4
- `switch` statements
  - `break`, 152–153
  - `case`, 151–152
  - `default`, 152–153
  - description of, 150–160
  - enumerated data types, 539–540
- symbols
  - `\` (backslash), 36, 38, 237
  - `{` (brace, left/opening brace), 29–31
  - `}`(brace, right/closing), 29, 31
  - `/** */` (comments, documentation), 80–82
  - `/* */` (comments, multi-line), 79–80
  - `//` (comments, single-line), 78–79
  - currency, 43, 48
  - `-`(decrement), 189–192
  - `<>` (diamond), 480
  - `/` (forward slash), 38, 237
  - `++` (increment), 189–192
  - `( )` (parentheses), 31, 59, 149, 272
  - `%` (percent), 59
  - `" "` (quotations, double), 31, 40, 41
  - `"` (quotations, single), 50
  - `;` (semicolons), 30–31, 117–118, 272
  - `//` (slash, double), 31
- symbols, escape sequences
  - `\\` (backslash, double), 37, 237
  - `\b` (backspace), 37
  - `\t` (horizontal tab), 37
  - `\n` (new line), 36–37
  - `\"` (quote, double), 37
  - `\'` (quote, single), 37
  - `\r` (return), 37
- symbols, operators
  - `=` (assignment), 52
  - `+=` (assignment), 64
  - `-=` (assignment), 64
  - `*=` (assignment), 64
  - `/=` (assignment), 64
  - `%=` (assignment), 64
- symbols operators (arithmetic)
  - `+` (addition), 55
  - associativity of, 58
  - `/` (division), 55, 56

  `%` (modulus), 55, 56
  `*` (multiplication), 55
  `-`(negation), 55
  precedence of operations, 57–59
  `+` (string concatenation), 40–41
  `-`(subtraction), 55
symbols, operators (logical)
  `&&` (AND), 134–137
  associativity of, 139–140
  `boolean` expressions using, 135
  `!` (NOT), 134, 139
  numeric ranges with, 140–141
  `||` (OR), 134, 137–139
  precedence of, 139–140
symbols, operators (relational)
  `==` (equal to), 113–114
  `>` (greater than), 113–114
  `>=` (greater than or equal to), 113–114
  `<` (less than), 113–114
  `<=` (less than or equal to), 113–114
  `!=` (not equal to), 113–114
synchronized methods, 589
syntax, 9
syntax errors, 12
`System` class
  `exit` method, 95–96
  `out` object, 33–34
`System.exit()`, 95–96
`System.out.printf()`, 160–174

**T**
tags
  `@exception`, 731–732
  `@param`, 288–290
  `@return`, 296–297
telephone numbers, formatting and unformatting example, 584–588
terminal operation, 1172
ternary operators, 55, 149
`TestScoreReader` class example, 597–600
`TextArea` controls, 885–887
`Text` class, 933–935
text editor, 12
`TextField` control, 762, 763
  reading input with, 799–803
text files, 230
`this` reference variable, 528
  calling overloaded constructor with, 530
  overcoming shadowing with, 529–531
thread, 95
three-dimensional arrays, 462
throwing exceptions, 725–732
`throws` clause, 233, 241, 724–726, 730
`throw` statement, 725–732
time sharing, 5
`toArray()`, 1179
`toBinaryString()`, 594–595
`toCharArray()`, 573–575
`toHexString()`, 594–595
tokenizing strings, 589–593
tokens, 589–593
`toList()`, 1185
`toLowerCase()`, 75, 563
`toMap()`, 1185
`toOctalString()`, 594–595
`toSet()`, 1185
`toString()`, 476, 505–507, 584, 594, 654
`toUpperCase()`, 75, 563
Towers of Hanoi, 1017–1021
trailing `else` clause, 131–132
trailing whitespace, 577, 592–593
transition classes, 940–941
  `FadeTransition` class, 954–955
  `FillTransition` class, 953–954
  `RotateTransition` class, 944–948
  `StrokeTransition` class, 952–953
  `TranslateTransition` class, 941–944
`TranslateTransition` class, 941–944
  `setByX` method, 942
  `setByY` method, 942
  `setFromX` method, 941
  `setFromY` method, 941
  `setToX` method, 941–942
  `setToY` method, 941–942
`TreeMap` class, 1159, 1168–1169
  constructors, 1169
`TreeSet` class, 1153–1158
  `comparator`, using with, 1156–1158
  constructors, 1158
`trim()`, 576, 577
truncated, 57, 66
truth tables
  NOT operator, 139
  AND operator, 132
  OR operator, 137–139
try block, 704
`try` statement
  catch block, 704
  catch clause, 704
  description of, 703–707
  exception handling, 717–718
  finally block, 719
  finally clause, 718–719
  try block, 704
two-dimensional arrays
  declaring, 450–452
  defined, 450
  displaying elements in, 457
  initializing, 454–456
  length field in, 455–457
  passing, to methods, 459–461
  ragged, 461
  summing columns in, 458–460
  summing elements in, 457–459
  summing rows in, 458
type parameter, 1084–1085, 1090, 1094–1095, 1104–1106
type wildcard, 1091–1092

**U**
unary operators, 55, 1126
unboxing, 595–596, 1087–1088
unchecked exceptions, 724–725
Unicode, 51–52, 119
Unicode Text Format (UTF)-8 encoding, 738
Unified Modeling Language (UML) diagrams
  abstract classes and methods, 666
  access specification, 339
  aggregation, 523
  `BankAccount class`, 372
  `Car` class, 387

Unified Modeling Language (UML) diagrams *(continued)*
  chains of inheritance, 652
  class design, 325
  constructors, 348
  `Customer` class, 386
  data type notation, 339–340
  inheritance, 620–621
  interfaces, 673–674
  parameter variable notation, 339–340
  protected members, 646
  `ServiceQuote` class, 388
  `Stock` class, 506
uniform resource locator, 970
uninitialized reference variable, 348
UNIX, 5
`UnsupportedOperationException`, 1159
upper bounds, generics, 1095
uppercase, converting characters to, 558
USB drives, 5
user-controlled loops, 206, 212–213
UTF (Unicode Text Format)-8 encoding, 738

**V**

validation, input, 200–203
`valueOf()`, 577–578
value-returning method, 269–271, 293–297
  `boolean` value, returning, 300
`values` method, 1163
vararg parameter, 470–472
variable declaration, 71
variable length argument lists, 470–472
variables
  accessing, within lambda expression, 688
  accumulators, 217, 219, 427
  assignment, 52–53
  class type, 71–72, 322
  counter, 208
  declaration, 39, 147–148
  defined, 11–12
  holding one value at a time, 53–54
  how to use, 39–44
  initialization, 52–53
  instance, 345
  lifetime, 292
  local, 76, 290–292
  names, 43
  parameter, 279, 281–282
  primitive, 321–324
  primitive type, 71–72
  reference, 72, 322–323, 348, 419–421, 528–530
  scope, 147–148
  scope of, 76–78, 282
  shadowing, 379, 529–531
  `this`, 528–530
`VBox` layout container, 782–784
vertex, 928
view (rooted tree), 1343
Visual Basic, 7
`void` methods, 270–272

**W**

`while` loop
  body of, 194
  braces, 197
  control variable, 196
  defined, 193
  `do-`, 204–206
  for input validation, 200–203
  header of, 194
  infinite, 196–197
  iterations, 195–196
  as pretest loop, 196
  program design with, 198–199
  programming style, 198
whitespace, 80, 577, 592–593
whole-part relationship, 529
widening conversion, 66
wildcard `import` statement, 381
wildcard, type, 1091–1092
Windows, operating system, 5
word processing, 5
worst case complexity function f(n), 1066–1067
wrapper classes
  autoboxing, 595–596
  `Byte`, 96, 594
  `Character`, 558–565
  defined, 557
  `Double`, 96, 594
  `Float`, 96, 594
  `Integer`, 96, 594
  `Long`, 96, 594
  `MAX_VALUE` constants, 595
  `MIN_VALUE` constants, 595
  numeric data types, 594–596
  parse methods, 96, 557, 594
  `Short`, 96, 594
  `substrings`, 566–575
  `toBinaryString()`, 594–595
  `toHexString()`, 594–595
  `toOctalString()`, 594–595
  `toString()`, 594
  unboxing, 595–596
`writeBoolean()`, 734
`writeByte()`, 734
`writeChar()`, 734
`writeDouble()`, 734
`writeFloat()`, 734
`writeInt()`, 734
`writeLong()`, 734
`writeObject()`, 745
`writeShort()`, 734
`writeUTF()`, 734
writing
  classes, step-by-step instructions, 324–340
  data to files, 733–735
  `RandomAccessFile` class and, 740–741
  strings, 738

**Z**

zeros, padding numbers with leading, 168–169

APPENDIX

# A The ASCII/Unicode Characters

The following table lists the first 127 Unicode character codes, which are the same as the ASCII (American Standard Code for Information Interchange) character set. This group of character codes is known as the *Latin Subset of Unicode*. The code columns show character codes and the character columns show the corresponding characters. For example, the code 65 represents the letter A. Note that the first 31 codes and code 127 represent control characters that are not printable.

| Code | Character | Code | Character | Code | Character | Code | Character | Code | Character |
|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | 26 | SUB | 52 | 4 | 78 | N | 104 | h |
| 1 | SOH | 27 | Escape | 53 | 5 | 79 | O | 105 | i |
| 2 | STX | 28 | FS | 54 | 6 | 80 | P | 106 | j |
| 3 | ETX | 29 | GS | 55 | 7 | 81 | Q | 107 | k |
| 4 | EOT | 30 | RS | 56 | 8 | 82 | R | 108 | l |
| 5 | ENQ | 31 | US | 57 | 9 | 83 | S | 109 | m |
| 6 | ACK | 32 | (Space) | 58 | : | 84 | T | 110 | n |
| 7 | BEL | 33 | ! | 59 | ; | 85 | U | 111 | o |
| 8 | Backspace | 34 | " | 60 | < | 86 | V | 112 | p |
| 9 | HTab | 35 | # | 61 | = | 87 | W | 113 | q |
| 10 | Line Feed | 36 | $ | 62 | > | 88 | X | 114 | r |
| 11 | VTab | 37 | % | 63 | ? | 89 | Y | 115 | s |
| 12 | Form Feed | 38 | & | 64 | @ | 90 | Z | 116 | t |
| 13 | CR | 39 | ' | 65 | A | 91 | [ | 117 | u |
| 14 | SO | 40 | ( | 66 | B | 92 | \ | 118 | v |
| 15 | SI | 41 | ) | 67 | C | 93 | ] | 119 | w |
| 16 | DLE | 42 | * | 68 | D | 94 | ^ | 120 | x |
| 17 | DC1 | 43 | + | 69 | E | 95 | _ | 121 | y |
| 18 | DC2 | 44 | ' | 70 | F | 96 | ` | 122 | z |
| 19 | DC3 | 45 | - | 71 | G | 97 | a | 123 | { |
| 20 | DC4 | 46 | . | 72 | H | 98 | b | 124 | \| |
| 21 | NAK | 47 | / | 73 | I | 99 | c | 125 | } |
| 22 | SYN | 48 | 0 | 74 | J | 100 | d | 126 | ~ |
| 23 | ETB | 49 | 1 | 75 | K | 101 | e | 127 | DEL |
| 24 | CAN | 50 | 2 | 76 | L | 102 | f | | |
| 25 | EM | 51 | 3 | 77 | M | 103 | g | | |

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

APPENDIX

# B

# Operator Precedence and Associativity

This table shows the precedence and associativity of all the Java operators. The table is divided into groups, and each operator in a group has the same precedence. The groups of operators are arranged from the highest precedence at the top of the table to the lowest precedence at the bottom of the table. For example, the first group of operators shown is:

```
. [] () ++ --
```

This group of operators has the highest precedence of all the operators; and each of these operators has the same precedence.

| Operator | Description | Associativity |
|---|---|---|
| `.` | membership | left-to-right |
| `[]` | array subscript | left-to-right |
| `()` | method argument list | left-to-right |
| `++` | postfix increment | left-to-right |
| `--` | postfix decrement | left-to-right |
| `++` | prefix increment | right-to-left |
| `--` | prefix decrement | right-to-left |
| `+` | unary plus | right-to-left |
| `-` | unary minus | right-to-left |
| `~` | bitwise complement | right-to-left |
| `!` | logical NOT | right-to-left |
| `new` | object creation | right-to-left |
| ***`(type)`*** | cast | right-to-left |
| `*` | multiplication | left-to-right |
| `/` | division | left-to-right |
| `%` | remainder | left-to-right |

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

| Operator | Description | Associativity |
|---|---|---|
| `+` | addition | left-to-right |
| `+` | string concatenation | left-to-right |
| `-` | subtraction | left-to-right |
| `<<` | left shift | left-to-right |
| `>>` | signed right shift | left-to-right |
| `>>>` | unsigned right shift | left-to-right |
| `<` | less than | left-to-right |
| `>` | greater than | left-to-right |
| `<=` | less than or equal to | left-to-right |
| `>=` | greater than or equal to | left-to-right |
| `instanceof` | type comparison | left-to-right |
| `==` | equal to | left-to-right |
| `!=` | not equal to | left-to-right |
| `&` | bitwise AND | left-to-right |
| `^` | bitwise XOR | left-to-right |
| `\|` | bitwise OR | left-to-right |
| `&&` | logical AND | left-to-right |
| `\|\|` | logical OR | left-to-right |
| `?:` | conditional | right-to-left |
| `=` | assignment | right-to-left |
| `+=` | combined assignment | right-to-left |
| `-=` | combined assignment | right-to-left |
| `*=` | combined assignment | right-to-left |
| `/=` | combined assignment | right-to-left |
| `<<=` | combined assignment | right-to-left |
| `>>=` | combined assignment | right-to-left |
| `>>>=` | combined assignment | right-to-left |
| `&=` | combined assignment | right-to-left |
| `^=` | combined assignment | right-to-left |
| `\|=` | combined assignment | right-to-left |

## APPENDIX C

# Java Key Words

The following words have reserved meaning in the Java language.

`abstract`
`assert`
`boolean`
`break`
`byte`
`case`
`catch`
`char`
`class`
`const`
`continue`
`default`
`do`
`double`
`else`
`enum`
`extends`
`false`
`final`
`finally`
`float`
`for`
`goto`
`if`
`implements`
`import`
`instanceof`
`int`
`interface`
`long`
`native`
`new`
`null`
`package`
`private`
`protected`
`public`
`return`
`short`
`static`
`strictfp`
`super`
`switch`
`synchronized`
`this`
`throw`
`throws`
`transient`
`true`
`try`
`void`
`volatile`
`while`

Note that the words `const` and `goto` are reserved, but they are not used.

The following words were introduced as *restricted words* in Java 9. They are treated as key words only within a module descriptor.

`exports`
`module`
`open`
`opens`
`provides`
`requires`
`to`
`transitive`
`uses`
`with`

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

# APPENDIX D

# Installing the JDK and Using the JDK Documentation

To write and execute Java programs, you will need to download and install the Java Development Kit (JDK). For this text, you need the JDK Standard Edition, which is available for the Windows, Linux, macOS, and Solaris operating systems. It can be downloaded from this website:

```
www.oracle.com/technetwork/java/javase/downloads/index.html
```

**NOTE:** If you would rather not type this entire address into your browser, you can also go to `www.oracle.com`, click *Downloads*, then click *Java for Developers*.

On the webpage, click the link to download the JDK. You will need to accept the license agreement, then select the version of the JDK for your operating system to begin the download.

Once the file is downloaded, execute it to start the installation. The installation program will install two items to your system: the JDK and the Java Runtime Environment (JRE). First the JDK will be installed. Click the Next button on each screen to accept the default selections, and be sure to take note of the location on your system where the JDK will be installed. Depending on the version of the JDK that you are installing, on windows the location will be something similar to

```
C:\Program Files\Java\jdk-9.0.1
```

Once the JDK is installed, the installation process for the JRE begins. Click the Next button on each screen to accept the default selections, and be sure to take note of the location on your system where the JRE will be installed. Depending on the version that you are installing, on windows the location will be something similar to

```
C:\Program Files\Java\jre-9.0.1
```

Access online material for this text at www.pearson.com/gaddis

## Setting the `Path` Environment Variable

If you plan to use the JDK command-line utilities to compile and run your programs, you will probably want to edit the contents of the `Path` variable on your system. This allows your system to find the JDK utilities from any folder when you run them at the command-line.
The `Path` variable contains a list of directory paths separated by semicolons. For example, on windows the `Path` variable might contain something like the following:

```
C:\Games;C:\Temp;C:\Program Files\MyPrograms
```

When you type the name of an executable file at the command-line and press Enter, the system will first look in the current folder for that file. If it cannot find the file there, it will begin looking in the folders that are listed in the `Path` variable. (On an actual system, the `Path` variable will contain many more paths than shown in this example, but this gives you an idea of how it works.)

We mentioned earlier that during the JDK installation process, you should take note of the location on your system where the JDK is installed. Inside that folder, there is another folder named `bin` that contains the JDK utility programs. The path to that folder on windows will be something like

```
C:\Program Files\Java\jdk-9.0.1\bin
```

**NOTE:** Keep in mind that the actual path on your system might differ slightly from this example, depending on the version of the JDK that you have installed.

To make it easy to execute the JDK utilities from the Windows command line, you should add this path to the `Path` variable. The procedure for adding this path to the `Path` variable depends on the version of Windows you are using. The steps required for Windows 10, Windows 8, and Windows 7 follow.

### Windows 10

In the Windows search box, type "environment variables" then select **Edit the system environment variables** from the search results.
This will display the *System Properties* window (with the *Advanced* tab selected at the top of the window). Click the Environment Variables ... button. Under **System Variables,** find *Path*, click on it, then click the Edit button. Add a semicolon to the end of the existing contents then add the path of the JDK utility programs. Click the OK buttons until all the dialog boxes are closed and exit the control panel.

### Windows 8

In the right bottom corner of the screen, click on the **Search** icon and type **Control Panel.** Click on **Control Panel,** then click **System,** then click **Advanced system settings.** Click on the **Advanced** tab, then click Environment Variables. Under **System Variables,** find Path, click on it, then click the Edit button. Add a semicolon to the end of the existing contents and then add the path of the JDK utility programs. Click the OK buttons until all the dialog boxes are closed and exit the control panel.

### Windows 7

Click the Start button then right-click **Computer**. On the pop-up menu, select **Properties**. In the window that appears next, click **Advanced system settings**. This displays the System Properties window. Click the Environment Variables ... button. In the System Variables list, scroll to the `Path` variable. Select the Path variable and click the Edit button. Add a semicolon to the end of the existing contents, and then add the path of the JDK utility programs. Click the OK buttons until all the dialog boxes are closed and exit the control panel.

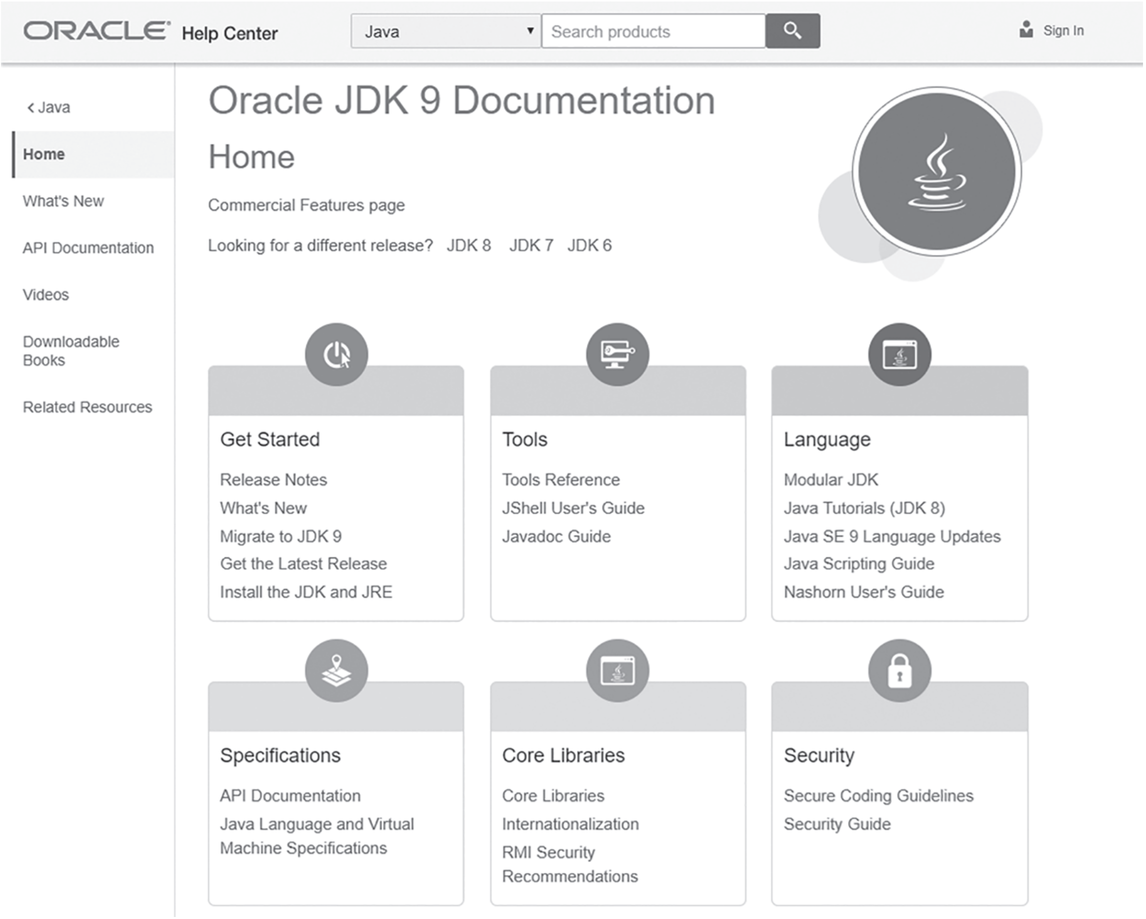
## Using the Java API Documentation

To view the official Java API documentation, go to the following site:

`http://docs.oracle.com/javase`

Figure D-1 shows how the page appears at the time this was written. The page provides links to several useful sites, including the official *API documentation*. As a student learning Java, or as a professional Java programmer, you will find yourself referring to the API documentation frequently.

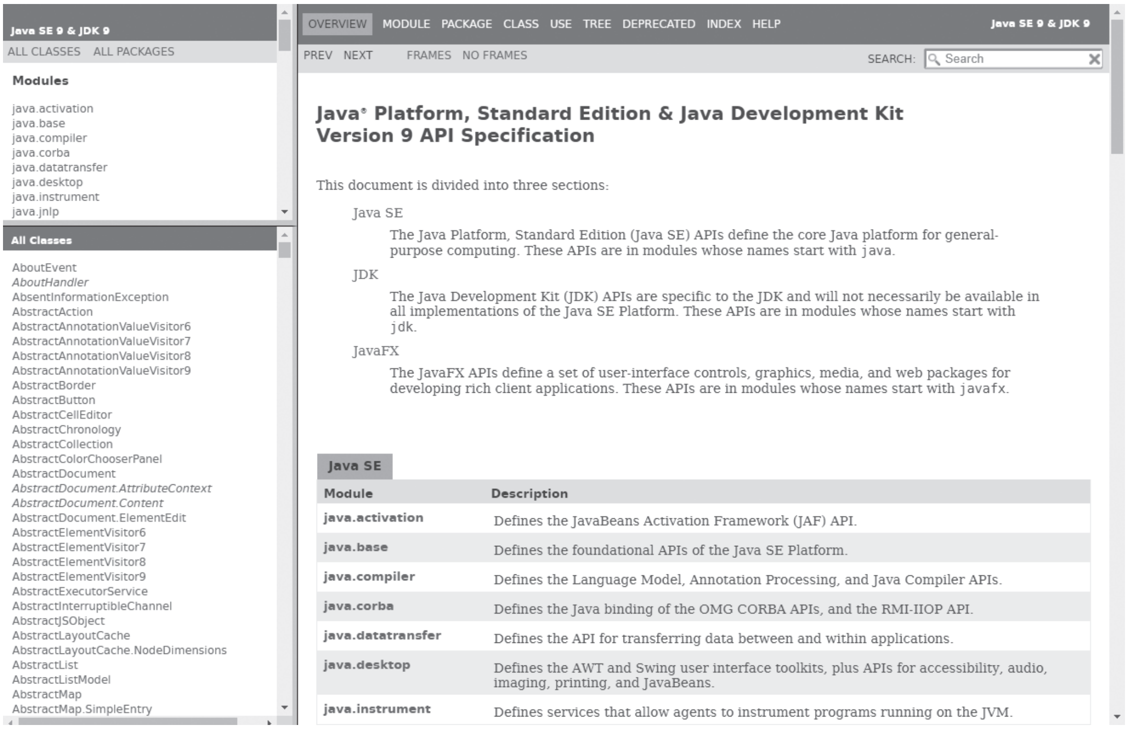
**Figure D-1** The Java SE documentation site (Oracle Corporate Counsel)



ORACLE Help Center
Java
Search products
Sign In
< Java
Home
What's New
API Documentation
Videos
Downloadable Books
Related Resources
Oracle JDK 9 Documentation
Home
Commercial Features page
Looking for a different release? JDK 8 JDK 7 JDK 6
Get Started
Release Notes
What's New
Migrate to JDK 9
Get the Latest Release
Install the JDK and JRE
Tools
Tools Reference
JShell User's Guide
Javadoc Guide
Language
Modular JDK
Java Tutorials (JDK 8)
Java SE 9 Language Updates
Java Scripting Guide
Nashorn User's Guide
Specifications
API Documentation
Java Language and Virtual Machine Specifications
Core Libraries
Core Libraries
Internationalization
RMI Security Recommendations
Security
Secure Coding Guidelines
Security Guide

When you click the link for the *API documentation*, and then click the link for *Frames*, you will see a page similar to the one shown in Figure D-2. In the leftmost pane, you will see an alphabetically ordered list of all the classes and interfaces in the Java API. To see the documentation for a particular class, simply scroll to the name of the class in the left pane, and click it.

**Figure D-2** Java SE API documentation (Oracle Corporate Counsel)



**Modules**

java.activation
java.base
java.compiler
java.corba
java.datatransfer
java.desktop
java.instrument
java.jnlp

**All Classes**

AboutEvent
*AboutHandler*
AbsentInformationException
AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractAnnotationValueVisitor8
AbstractAnnotationValueVisitor9
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractChronology
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
*AbstractDocument.AttributeContext*
*AbstractDocument.Content*
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractElementVisitor8
AbstractElementVisitor9
AbstractExecutorService
AbstractInterruptibleChannel
AbstractJSObject
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry

**Java® Platform, Standard Edition & Java Development Kit Version 9 API Specification**

This document is divided into three sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

JavaFX

The JavaFX APIs define a set of user-interface controls, graphics, media, and web packages for developing rich client applications. These APIs are in modules whose names start with `javafx`.

**Java SE**

| Module | Description |
|---|---|
| **java.activation** | Defines the JavaBeans Activation Framework (JAF) API. |
| **java.base** | Defines the foundational APIs of the Java SE Platform. |
| **java.compiler** | Defines the Language Model, Annotation Processing, and Java Compiler APIs. |
| **java.corba** | Defines the Java binding of the OMG CORBA APIs, and the RMI-IIOP API. |
| **java.datatransfer** | Defines the API for transferring data between and within applications. |
| **java.desktop** | Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans. |
| **java.instrument** | Defines services that allow agents to instrument programs running on the JVM. |

APPENDIX

# E

# Using the javadoc Utility

The Java JDK comes with a utility named `javadoc`, which you can use to automatically generate documentation for the classes, interfaces, and methods that you write. The `javadoc` utility produces HTML documentation with the same structure as the Java API documentation. To generate this documentation, there are two general steps necessary:

1. Write documentation comments in the Java source code file that you wish to document.
2. Run the `javadoc` utility, passing the name of the Java source code file as a command-line argument.

Let's take a closer look at each step.

## Writing Documentation Comments

In the first step, you write into a Java source file special comments known as *documentation comments*. You typically do this as you are writing the Java code. A documentation comment may precede a class header, an interface header, or a method header. Each one contains information about the class, interface, or method that it precedes.

A documentation comment begins with `/**` and ends with `*/`. Between these two symbols you write information that the `javadoc` utility can use to generate documentation. This information can contain regular text descriptions, as well as special `javadoc` tags which begin with the `@` symbol. When you run the `javadoc` utility, it searches for these documentation comments and processes them.

To see examples of documentation comments, we will look at the `BankAccount` class that was first presented in Chapter 6. This class uses simple documentation comments for the class and its methods. First, here is the documentation for the class, followed by the class header:

```
/**
  The BankAccount class simulates a bank account.
*/

public class BankAccount
```

Access online material for this text at www.pearson.com/gaddis

The text in a documentation comment for a class should contain a brief description of the class. This is the text that will be used to describe the class in the HTML file. Notice in this example, the description inside the documentation comment is indented with spaces. Any whitespace characters at the beginning of a line are ignored by the `javadoc` utility. In addition to whitespace characters, `javadoc` ignores all asterisk characters at the beginning of a line. For example, the following documentation comment will produce the same results as the previous one:

```
/**
 *
 * The BankAccount class simulates a bank account.
 *
 */
public class BankAccount
```

Although the extra asterisks are ignored by `javadoc`, they help the comment to visually stand out for any person reading the code.

A documentation comment for a method is structured in the following way:

- After the `/**` symbol, a brief description of the method appears. This text will be used to describe the method in the HTML documentation file. In addition, the first sentence of this description will be used as a summary of the method.
- If the method has parameters, documentation for each parameter will appear in the comment. A parameter's documentation begins with the `@param` tag, followed by the name of the parameter, followed by a description of the parameter.
- If the method returns a value, a description of the return value will appear in the comment. This description must begin with the `@return` tag, followed by a description of the return value.
- If the method throws exceptions, documentation for each exception will appear in the comment. Documentation for each exception begins with the `@exception` tag, followed by the name of the exception, followed by a description of the events that cause the exception.

For example, here is one of the `BankAccount` constructors, preceded by its documentation comment:

```
/**
 * The constructor initializes the balance
 * and interestRate fields.
 * @param startBalance The starting balance
 * @param intRate The interest rate
 */
public BankAccount(double startBalance,
                   double intRate)
{
   balance = startBalance;
   interestRate = intRate;
   interest = 0.0;
}
```

The documentation comment for this constructor is relatively simple, containing only a brief one-sentence description and two `@param` tags.

In the HTML file that is produced by `javadoc`, each method will have two major sections of documentation: a *summary section* and a *detail section*. The first sentence in the method's documentation comment is used as the summary of the method. Note `javadoc` considers the end of the sentence as a period followed by a whitespace character. For this reason, when a method description contains more than one sentence, you should always end the first sentence with a period followed by a whitespace character. The method's detail section will contain all of the text from the beginning of the comment to the first tag, or the end of the comment if it contains no tags.

## The `@param` Tag

When the `javadoc` utility sees an `@param` tag inside of a method's documentation comments, it knows that the documentation for a parameter variable appears next. Each parameter should have its own `@param` tag, and follow this general format:

```
@param parameterName Description
```

In the general format, *`parameterName`* is the name of the parameter, and *`Description`* is a brief description of the parameter. When a method's documentation comments contain one or more `@param` tags, the `javadoc` utility will create a Parameters section in the method's documentation. This is where the descriptions of the method's parameters will be listed. Remember the following points about `@param` tag comments:

- All `@param` tags in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

## The `@return` Tag

Here is another method from the `BankAccount` class, which has a return value:

```
/**
 * The getBalance method returns the balance.
 * @return The value in the balance field.
 */
public double getBalance()
{
   return balance;
}
```

Because this method returns a value, the documentation comment contains an `@return` tag. The documentation for the return value follows this general format:

```
@return  Description
```

In the general format, *`Description`* is a brief description of the return value. When a method's documentation comments contain an `@return` tag, the `javadoc` utility will create a Returns section in the method's documentation. This is where the description of the method's return value will be listed. Remember the following points about `@return` tag comments:

- The `@return` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

## The `@exception` and `@throws` Tags

The `@exception` and `@throws` *tags* both do the same thing: specify an exception that is thrown by a method. Here are the general formats of both tags:

```
@exception  ExceptionName Description
@throws  ExceptionName Description
```

*`ExceptionName`* is the name of an exception and *`Description`* is a description of the circumstances that cause the exception. When a method's documentation comments contains an `@exception` tag or a `@throws` tag, the `javadoc` utility will create a Throws section in the method's documentation. This is where the description of the method's return value will be listed. Remember the following points about `@exception` and `@throws` tag comments:

- An `@exception` tag or `@throws` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

For example, the following code shows a method named `FileRead`, which throws an `IOException` when it fails to read a file:

```
/**
 * This method opens a file and reads its contents.
 * @param FileName Specifies the file to read.
 * @exception IOException Failed to read the file.
 */
public void FileRead(String filename) throws IOException
{
   // Body of the method appears here...
}
```

## Running `javadoc`

After creating one or more source code files containing documentation comments, your next step is to run the `javadoc` utility. You run `javadoc` from the operating system command prompt. Here is the general format of the `javadoc` command:

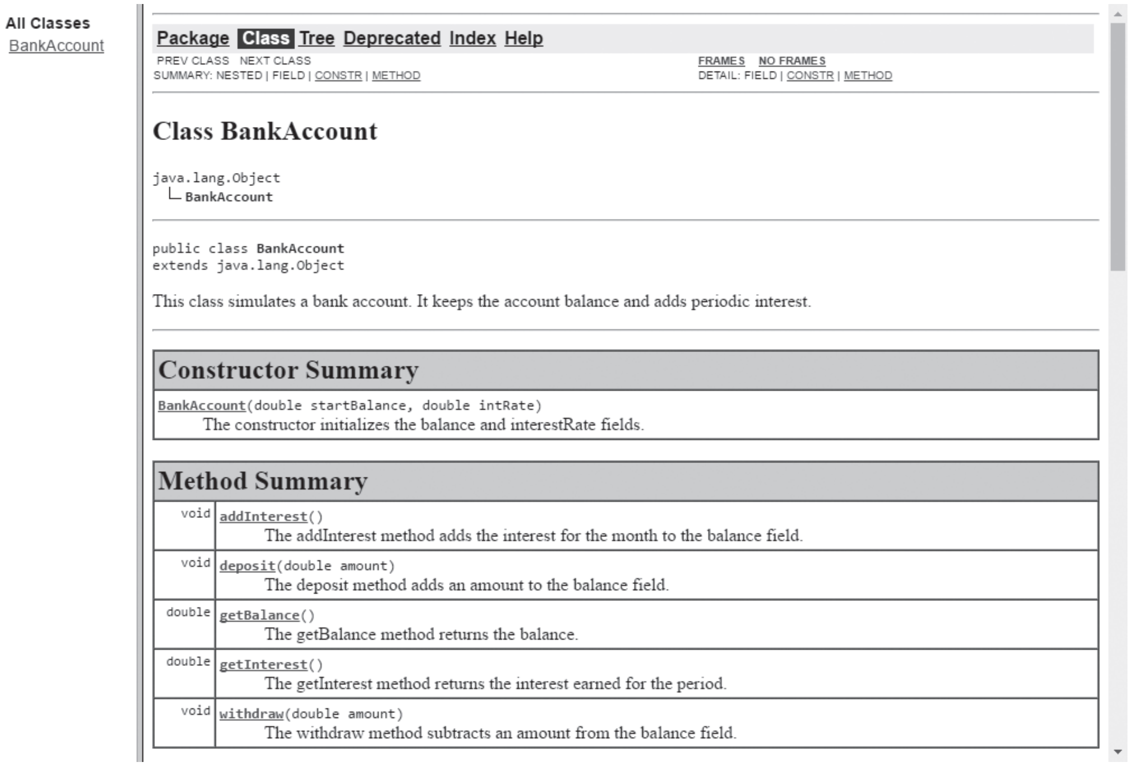
```
javadoc  SourceFileList
```

*`SourceFileList`* is one or more names of source code files. The files specified as arguments will be read by `javadoc` and documentation will be produced for each of them. For example, the following command will produce documentation for the `BankAccount` class:

```
javadoc BankAccount.java
```

After this command executes, several HTML files will be created in the same directory as the source code file. One of these files will have the same name as the class file. In this case, it will be named *BankAccount.html*. This is the file that you open in your browser to view the documentation. Figure E-1 shows the documentation for the `BankAccount` class.

**Figure E-1** `BankAccount` class documentation (Oracle Corporate Counsel)



All Classes
BankAccount
Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD
FRAMES NO FRAMES
DETAIL: FIELD | CONSTR | METHOD

Class BankAccount

```
java.lang.Object
  └ BankAccount
```

```
public class BankAccount
extends java.lang.Object
```

This class simulates a bank account. It keeps the account balance and adds periodic interest.

Constructor Summary

`BankAccount(double startBalance, double intRate)`
The constructor initializes the balance and interestRate fields.

Method Summary

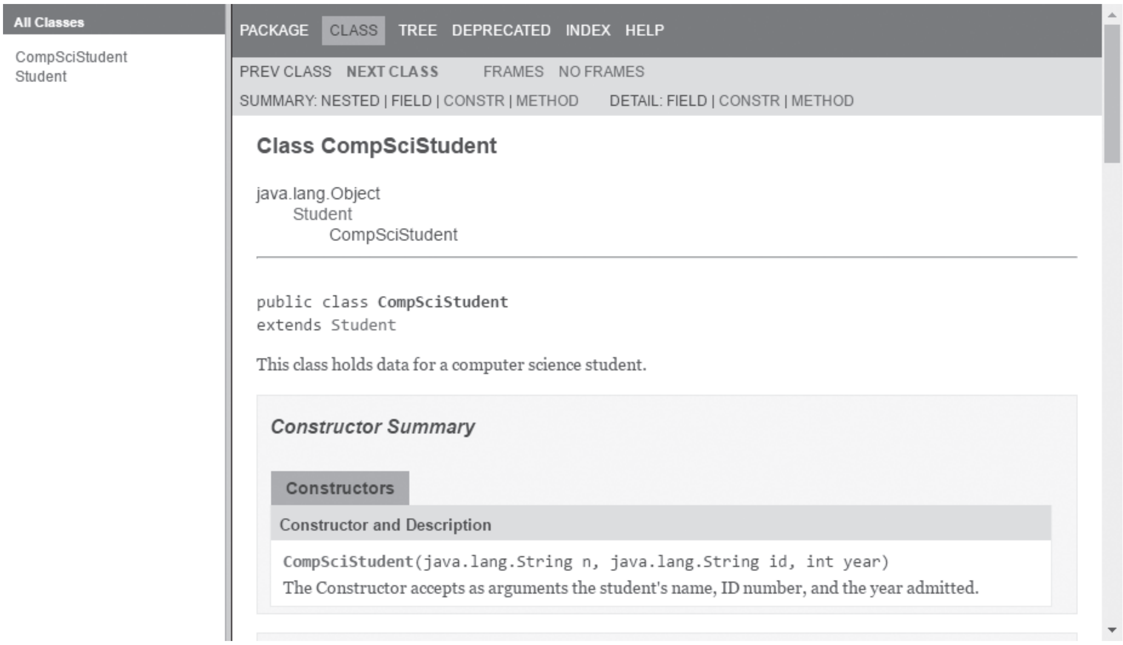
| | |
|---|---|
| `void` | `addInterest()`<br>The addInterest method adds the interest for the month to the balance field. |
| `void` | `deposit(double amount)`<br>The deposit method adds an amount to the balance field. |
| `double` | `getBalance()`<br>The getBalance method returns the balance. |
| `double` | `getInterest()`<br>The getInterest method returns the interest earned for the period. |
| `void` | `withdraw(double amount)`<br>The withdraw method subtracts an amount from the balance field. |

If you want to produce documentation for multiple source code files, simply separate the names of the files with spaces. Here is an example:

```
javadoc Student.java CompSciStudent.java
```

After this command executes, the resulting HTML files will contain documentation for the `Student` and `CompSciStudent` classes. One of the HTML files will be named *index.html*. You can open this file in your browser to view the documentation for each class. Figure E-2 shows an example. This document displays a frame on the left side of the screen containing links to each class's documentation.

**Figure E-2** A documentation file for multiple classes (Oracle Corporate Counsel)



**All Classes**

CompSciStudent
Student

PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

**Class CompSciStudent**

java.lang.Object
    Student
        CompSciStudent

```
public class CompSciStudent
extends Student
```

This class holds data for a computer science student.

*Constructor Summary*

**Constructors**

| Constructor and Description |
|---|
| `CompSciStudent(java.lang.String n, java.lang.String id, int year)`<br>The Constructor accepts as arguments the student's name, ID number, and the year admitted. |

## The `@author` and `@version` Tags

The `@author` tag may be used in a class's documentation comment to identify the author or authors of the class. The general format of the tag is:

```
@author  AuthorName
```

If there is more than one author, you use a separate tag for each author.

The `@version` tag may also be used in a class's documentation comment to identify the version of the class. The general format of the tag is:

```
@version  VersionNumber
```

Here is an example of the `BankAccount` class's documentation comment, modified to use these tags:

```
/**
 * The BankAccount class simulates a bank account.
 * @author Herbert Dorfmann
 * @version 1.0
 */

public class BankAccount
```

When you use these tags, you must also provide the `–author` and `–version` options on the command line when invoking the `javadoc` utility. Here is an example:

```
javadoc –author –version BankAccount.java
```

This will cause the resulting HTML documentation to contain the author name(s) and the version number. If you do not provide these command line options, `javadoc` will ignore the `@author` and `@version` tags.

## Embedding HTML in Documentation Comments

You can embed HTML tags in documentation comments and `javadoc` will include those tags when it generates the documentation files. For example, the following code shows a method with the  and  tags embedded. The  tag causes its enclosed text to be displayed in code font, and the  tag causes its enclosed text to be displayed in italics.

```
/**
 * The  addInterest  method adds the interest
 * for the  month  to the  balance  field.
 */

public void addInterest()
{
   interest = balance * interestRate;
   balance = balance + interest;
}
```

## The `-public`, `-private`, and `-protected` Command Line Options

By default, the `javadoc` utility generates documentation only for public and protected classes, interfaces, and members. You can alter this, however, by using a command line option.

The `-public` option causes `javadoc` to generate documentation only for public classes, interfaces, and members. Here is an example:

```
javadoc -public Class1.java Class2.java Class3.java
```

If any of the classes or any of the class members are not public, `javadoc` will not generate documentation for them.

The `-private` option causes `javadoc` to generate documentation for all classes, interfaces, and members. Here is an example:

```
javadoc -private Class1.java Class2.java Class3.java
```

This command will generate documentation for all the classes and their members, regardless of their access specification.

The `-protected` option causes `javadoc` to generate documentation only for public and protected classes, interfaces, and members. This is the same as `javadoc`'s default behavior. Here is an example:

```
javadoc -protected Class1.java Class2.java Class3.java
```

APPENDIX

# F

# More about the Math Class

The Java `Math` class is a collection of static methods for performing specific mathematical operations. This class is in the `java.lang` package, so there is no need for an `import` statement to use it.

In Chapter 2, you were introduced to the `Math.pow` method, which returns the value of a number raised to a power. Table F-1 describes several of the `Math` class's methods.

**Table F-1** Several `Math` class methods

| Method | Example Usage | Description |
|---|---|---|
| `abs` | `y = Math.abs(x);` | Returns the absolute value of the argument. This method can accept and return values of the `double`, `float`, `int`, and `long` data types. |
| `acos` | `y = Math.acos(x);` | Returns the arc-cosine of the argument. The argument should be the cosine of an angle. (The argument's value must be in the range from −1.0 through 1.0.) The return type and the argument are `double`s. |
| `asin` | `y = Math.asin(x);` | Returns the arc-sine of the argument. The argument should be the sine of an angle. (The argument's value must be in the range from −1.0 through 1.0.) The return type and the argument are `double`s. |
| `atan` | `y = Math.atan(x);` | Returns the arc-tangent of the argument. The argument should be the tangent of an angle. The return type and the argument are `double`s. |
| `cbrt` | `y = Math.cbrt(x);` | Returns the cube root of the argument. The return type and argument are `double`s. Note: This method was introduced in Java 5. |

Access online material for this text at www.pearson.com/gaddis

| Method | Example Usage | Description |
|---|---|---|
| `ceil` | `y = Math.ceil(x);` | Returns the smallest number that is greater than or equal to the argument. The return type and the argument are `double`s. |
| `cos` | `y = Math.cos(x);` | Returns the cosine of the argument. The argument should be an angle expressed in radians. The return type and the argument are `double`s. |
| `exp` | `y = Math.exp(x);` | Computes the exponential function of the argument, which is $e^x$. The return type and the argument are `double`s. |
| `floor` | `y = Math.floor(x);` | Returns the largest number that is less than or equal to the argument. The return type and the argument are `double`s. |
| `hypot` | `c = Math.hypot(a, b);` | Returns the length of the hypotenuse of a right triangle. The arguments `a` and `b` are the lengths of the other two sides of the triangle. The return type and the argument are `double`s. Note: This method was introduced in Java 5. |
| `log` | `y = Math.log(x);` | Returns the natural logarithm of the argument. The return type and the argument are `double`s. |
| `pow` | `y = Math.pow(x, z);` | Returns the value of the first argument raised to the power of the second argument. The return type and the argument are `double`s. |
| `round` | `y = Math.round(x);` | Returns the value of the argument, as an integer, rounded to the nearest whole number. The argument is expected to be a `double` or a `float`. If the argument is a `double`, the return type is `long`. If the argument is a `float`, the return type is `int`. |
| `sin` | `y = Math.sin(x);` | Returns the sine of the argument. The argument should be an angle expressed in radians. The return type and the argument are `double`s. |
| `sqrt` | `y = Math.sqrt(x);` | Returns the square root of the argument. The return type and argument are `double`s. |
| `tan` | `y = Math.tan(x);` | Returns the tangent of the argument. The argument should be an angle expressed in radians. The return type and the argument are `double`s. |
| `toDegrees` | `y = Math.toDegrees(x);` | Accepts as an argument an angle in radians. The angle converted to degrees is returned. The argument and return values are both `double`s. |
| `toRadians` | `y = Math.toRadians(x);` | Accepts as an argument an angle in degrees. The angle converted to radians is returned. The argument and return values are both `double`s. |

We will not cover all of these methods in depth, but let's take a closer look at some of them. The `Math.abs` method returns the absolute value of its argument. The following code segment shows an example of how it is used.

```
double x = -4.2, y = 4.2, a, b;
a = Math.abs(x);
b = Math.abs(y);
```

After this code executes, both the variables `a` and `b` will contain the value 4.2. The argument to the `Math.abs` method can be of the `double`, `float`, `int`, or `long` data types. This is because there are several overloaded versions of the method. The method's return value will be of the same data type as the argument.

Here is a program segment that demonstrates the `Math.sqrt` method, which returns the square root of a number:

```
num = 25.0;
s = Math.sqrt(num);
System.out.println("The square root of " + num + " is " + s);
```

The output of this program segment is

```
The square root of 25.0 is 5.0
```

The `Math.round` method returns the value of its argument rounded to the nearest whole number. The following code segment shows an example.

```
double x = 4.2, y = 4.8;
long a, b;
a = Math.round(x);
b = Math.round(y);
```

After this code executes, `a` will contain 4 and `b` will contain 5.

In addition to these and other methods, the `Math` class defines two static `final` variables, which are listed in Table F-2.

**Table F-2** The **E** and **PI** constants

| Method | Example Usage | Description |
|---|---|---|
| `E` | `y = x * Math.E;` | This is the mathematical constant known as e (for Euler's number). It is defined as 2.7182818284590452354. |
| `PI` | `area = Math.PI * radius * radius;` | This is the mathematical constant Pi, or π. It is defined as 3.14159265358979323849. |

Both of these `final` variables are public, so you can access them directly from the class.

APPENDIX

# G Packages

**NOTE:** To use this appendix, you must understand how your operating system uses directories, or folders. In addition, you must know how to set the value of an environment variable. The process of setting an environment variable differs among operating systems.

So far, you have been storing the classes you have created in the same folder or directory as the program that uses them. That is where the compiler looks for classes by default. In real-world application development, however, this approach is less than ideal. If you are developing more than one application that uses the same set of classes, you would have to make separate copies of the classes and store them in the same disk location as each application. A better approach is to have the classes stored in a central location available to all applications. Only one copy of the classes is needed, regardless of the number of applications you develop using them. In Java, this can be accomplished by using packages.

A *package*, which is also called a *library*, is a named group of related classes. Packages are stored in their own folder or directory. The compiler is informed of the package's location, so it can find it regardless of where the application that uses the package may be stored.

Let's look at a simple example. Suppose we create two classes: `Car` and `Truck`. Both classes are part of a package named `vehicles`. Code Listing G-1 shows the listing for the `Car` class, and Code Listing G-2 shows the listing for the `Truck` class.

Access online material for this text at www.pearson.com/gaddis

**Code Listing G-1** (Car.java)

```java
package vehicles;

/**
 * This class is in the vehicles package.
 */

public class Car
{
   private int passengers;   // Number of passengers
   private double topSpeed;  // Top speed in miles per hour

   /**
    * Constructor
    */

   public Car(int passengers, int topSpeed)
   {
      this.passengers = passengers;
      this.topSpeed = topSpeed;
   }

   /**
    * toString method
    */

   public String toString()
   {
      return "Passengers: " + passengers
           + "\nTop Speed: " + topSpeed + " miles per hour";
   }
}
```

**Code Listing G-2** (`Truck.java`)

```
package vehicles;

/**
 * This class is in the vehicles package.
 */

public class Truck
{
   private double mpg;  // Fuel economy in miles-per-gallon
   private double tons;  // Hauling capacity in tons

   /**
    * Constructor
    */

   public Truck(double mpg, double tons)
   {
      this.mpg = mpg;
      this.tons = tons;
   }

   /**
    * toString method
    */

   public String toString()
   {
      return "Fuel economy: " + mpg + " miles per gallon"
           + "\nHauling capacity: " + tons + " tons";
   }
}
```

Notice the first line of each file reads:

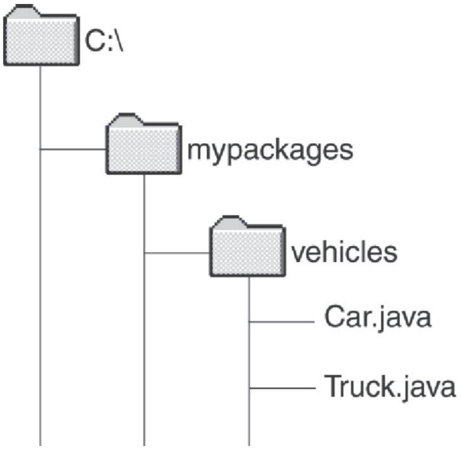
```
package vehicles;
```

The word `package` is a key word in Java, and `vehicles` is the name of the package to which the class belongs. Notice the statement is terminated by a semicolon. This statement informs the compiler that the contents of the file belong to the `vehicles` package.

> **NOTE:** The compiler expects the package statement to be the first statement in a file. Only comments and blank lines may be placed before it.

When a class is part of a package, the compiler expects the class file to be stored in a directory with the same name as the package. For example, the `Car` and `Truck` classes are part of the `vehicles` package, so they must be stored in a directory named *vehicles*. Typically, the *vehicles* directory would be stored under another directory that acts as the base directory for packages. Figure G-1 depicts an example directory structure on a Windows system. The *mypackages* directory is the base directory for packages. Under *mypackages*, the *vehicles* directory appears, which holds the *Car.java* and *Truck.java* files.

**Figure G-1** Example directory structure for storing a package on windows



Once a package has been created, the Java compiler must be informed of its location. The exact process that you use depends on your Java compiler and your operating system. You should consult your instructor for precise details, but we will give a general overview of what needs to be done when using the Oracle JDK under Windows and UNIX or Linux.

The Oracle JDK uses an operating system environment variable named `CLASSPATH` to determine where the base directory for your packages is located. The `CLASSPATH` environment variable holds a string that consists of one or more pathnames, separated by a delimiting character. To inform the compiler of your base directory's location, simply add its pathname to the contents of the `CLASSPATH` environment variable.

## Setting `CLASSPATH` under Windows

The following is an example command that can be used at the command line in Windows to set the `CLASSPATH` environment variable:

```
set CLASSPATH=C:\mypackages;
```

There are no spaces before or after the = sign. In Windows, multiple pathnames stored in the variable are separated by a semicolon. This command stores two pathnames in the `CLASSPATH` environment variable: `C:\mypackages` and the . symbol, which represents the current directory. This tells Java to search two locations for packages: `C:\mypackages` and the current directory. Here is another example:

```
set CLASSPATH=C:\mypackages;C:\project;
```

This command stores three pathnames in the `CLASSPATH` environment variable: `C:\myPackages`, `C:\project`, and the . symbol, which represents the current directory. This tells Java to search these three locations when looking for packages.

## Setting `CLASSPATH` under UNIX or Linux

**NOTE:** The UNIX and Linux instructions in this appendix assume you are using the `bash` shell.

The following is an example command that can be used in UNIX or Linux to set the `CLASSPATH` environment variable.

```
export CLASSPATH=/home/tsmith/mypackages:
```

In UNIX and Linux, multiple pathnames stored in the variable are separated by a colon. This command stores two pathnames in the `CLASSPATH` environment variable: `/home/tsmith/mypackages` and the . symbol, which represents the current directory. This tells Java to search two locations for packages: `/home/tsmith/mypackages` and the current directory. Here is another example:

```
export CLASSPATH=/home/tsmith/mypackages:/home/tsmith/project;
```

This command stores three pathnames in the `CLASSPATH` environment variable: `/home/tsmith/mypackages`, `/home/tsmith/project`, and the . symbol, which represents the current directory. This tells Java to search these three locations when looking for packages.

**NOTE:** You should always include the dot symbol, which represents the current directory, as a pathname in the `CLASSPATH` variable. If you do not, your programs will not run correctly.

## Using the `import` Statement

When a package has been created and stored on the disk as previously described and the `CLASSPATH` variable has been set with the pathname of the base directory for your packages, you are ready to create programs that use your package. Code Listing G-3 uses both the `Car` and `Truck` classes.

**Code Listing G-3** (`CarTruckDemo.java`)

```
 1  import vehicles.*;
 2
 3  /**
 4   * This program demonstrates the Car and Truck
 5   * classes which are part of the vehicles package.
 6   */
 7
 8  public class CarTruckDemo
 9  {
10     public static void main(String[] args)
11     {
12        Car roadster = new Car(2, 155);
13        Truck pickUp = new Truck(18, 2);
14
15        System.out.println("Here's information " +
16                           "about the car:");
17        System.out.println(roadster);
18        System.out.println();
19        System.out.println"Here's information " +
20                          "about the truck:");
21        System.out.println(pickUp);
22     }
23  }
```

**Program Output**

```
Here's information about the car:
Passengers: 2
Top speed: 155.0 miles per hour

Here's information about the truck:
Fuel economy: 18.0 miles per gallon
Hauling capacity: 2.0 tons
```

Notice line 1 contains an `import` statement. The word `import` is a key word in Java. The name that follows `import` is the name of a package the program intends to use. The `.*` that follows the package name means to import all the classes that are part of that package. This statement tells the compiler to make all the classes that are part of the `vehicles` package available to the program. If we only wanted to make the `Truck` class available, we could have used the following `import` statement:

```
import vehicles.Truck;
```

In this case, the compiler would only make the `Truck` class available. Any references to the `Car` class would cause an error. Likewise, we could use the following statement to make only the `Car` class available:

```
import vehicles.Car;
```

You will recall that you have previously used the following `import` statement in programs that use the `DecimalFormat` class:

```
import java.text.DecimalFormat;
```

This statement tells the compiler to use the `DecimalFormat` class, which is part of the `java.text` package. The `java.text` package is part of the Java standard class library. You have also used the following `import` statement in programs using the `Scanner` class:

```
import java.util.Scanner;
```

This statement tells the compiler to use the `Scanner` class, which is part of the `java.util` package. The `java.util` package is also part of the Java standard class library. At this point, it might be helpful to summarize the steps necessary to create and use a package:

1. Place an appropriate `package` statement in each class file that is to be part of the package. The `package` statement must be the first line of the file or preceded only by comments and/or blank lines.
2. You should have a base directory on your system for storing your packages. Under this directory, create a directory that bears the same name as the package. Store the package's class files in this folder.
3. Add the pathname of the package base directory to the `CLASSPATH` environment variable.
4. Place an appropriate `import` statement in each program that intends to use the package.

## Using Fully Qualified Class Names

It is not required that you use the `import` statement to access classes in a package. Without the `import` statement, however, you must use the classes' fully qualified names. The fully qualified names of the `Car` and `Truck` classes in the `vehicle` package are `vehicles.Car` and `vehicles.Truck`. Code Listing G-4 is a version of Code Listing G-3, which uses fully qualified class names instead of the `import` statement.

**Code Listing G-4** `(CarTruckDemo2.java)`

```
/**
 * This program demonstrates the Car and Truck
 * classes by using their fully-qualified names
 * instead of an import statement.
 */

public class CarTruckDemo2
{
   public static void main(String[] args)
   {
      vehicles.Car roadster = new vehicles.Car(2, 155);
      vehicles.Truck pickUp = new vehicles.Truck(18, 2);

      System.out.println("Here's information " +
                         "about the car:");
      System.out.println(roadster);
      System.out.println();
      System.out.println("Here's information " +
                         "about the truck:");
      System.out.println(pickUp);
   }
}
```

**Program Output**

Same as Code Listing G-3.

Notice every occurrence of the `Car` and `Truck` class names must be written as `vehicles.Car` and `vehicles.Truck`. Obviously, it is more convenient to use the `import` statement.

## Using Package Subdirectories

You have learned that the Java compiler locates packages by searching the directories listed by the `CLASSPATH` environment variable. It is possible to create subdirectories in these locations and store packages there as well. For example, suppose we create a set of classes for calculating the pay of employees. The first step would be to create a folder to hold the packages. Figure G-2 shows the `employees` directory has been added to the `mypackages` directory.

Under the `employees` directory are two other directories: `hourly` and `salaried`. Figure G-3 shows the contents of the `hourly` directory.

The files *Clerical.java* and *AssemblyLine.java* contain code for classes that calculate the hourly pay for clerical staff and assembly line workers. If we looked at the *Clerical.java* file, we would see something similar to the following:

```
package employees.hourly;
/**
 * This class is part of the employees.hourly package.
 */
public class Clerical
{
   Details of this class are omitted.
}
```

**Figure G-2** `employees` directory added to the `mypackages` directory

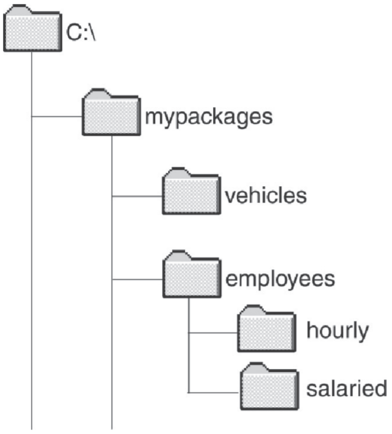
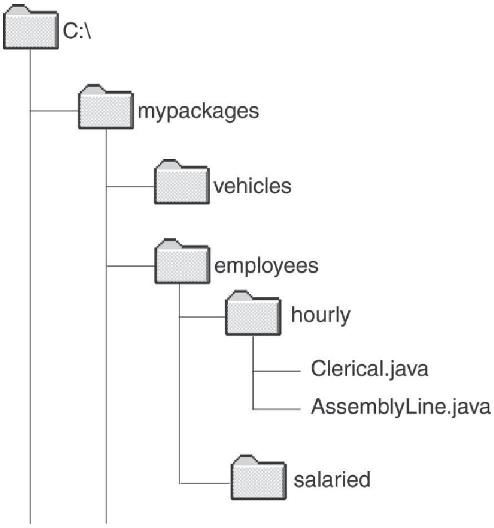

C:\
mypackages
vehicles
employees
hourly
salaried

**Figure G-3** Directory structure showing the contents of the `hourly` directory


C:\
mypackages
vehicles
employees
hourly
Clerical.java
AssemblyLine.java
salaried

If we looked in the *AssemblyLine.java* file, we would see something similar to the following:

```
package employees.hourly;
/**
 * This class is part of the employees.hourly package.
 */
public class AssemblyLine
{
   Details of this class are omitted.
}
```

Notice the first line of both files is the following `package` statement:

```
package employees.hourly;
```

The files are stored in the `hourly` directory. Because `hourly` is a subdirectory of `employees`, the two names are separated by a period. This makes the package name `employees.hourly`. The following `import` statement could then be added to any program that needs to use the `Clerical` or `AssemblyLine` classes:

```
import employees.hourly.*;
```

This statement makes all of the classes that are part of the `employees.hourly` package accessible. As before, individual classes can be specifically named in the `import` statement. For example, the following statement will make only the `Clerical` class accessible:

```
import employees.hourly.Clerical;
```

## Access Specifiers and Packages

So far you have learned that class members may be declared as either `public` or `private`. Members that are declared as `public` may be accessed by statements outside the class as well as inside the class. Members that are declared as `private`, however, may only be accessed by statements inside the class.

In addition to `public` and `private`, Java also allows class members to be declared with no access specifier at all. When a class member is declared with no access specifier, it is accessible by any statement within the same package. For example, consider the following scenario. Class `A` and class `B` are in the same package. The variable `x` is a member of class `A` and is declared with no access specifier. Because `x` is declared with no access specifier, the methods in class `B` have access to it (as if it were `public`). Statements that are outside the package, however, cannot access `x`. Table G-1 summarizes the effect of access specifiers, as they pertain to what you have learned so far.

**Table G-1** The effect of access specifiers

| Access Attribute | Description |
|---|---|
| No attribute | May be applied to classes and class members (variables or methods). This makes a class or class member accessible within the package. |
| `public` | May be applied to classes and class members (variables or methods). This makes a class or class member accessible to all statements in the program (inside or outside the package). |
| `private` | May only be applied to member variables and member methods. A private variable or method is only accessible by statements in the same class. |

## The Standard Java Packages

The standard Java classes that make up the API are organized into packages. Table G-2 lists a few of them.

**Table G-2** A few of the standard Java packages

| Package | Description |
|---|---|
| `java.applet` | Provides the classes necessary to create an applet. |
| `java.awt` | Provides classes for the Abstract Windowing Toolkit. These classes are used in drawing images and creating graphical user interfaces. |
| `java.io` | Provides classes that perform various types of input and output. |
| `java.lang` | Provides general classes for the Java language. This package is automatically imported. |
| `java.net` | Provides classes for network communications. |
| `java.security` | Provides classes that implement security features. |
| `java.sql` | Provides classes for accessing databases using structured query language. |
| `java.text` | Provides various classes for formatting text. |
| `java.util` | Provides various utility classes. |
| `javax.swing` | Provides classes for creating graphical user interfaces. |

To use a class from a Java package, you must have an appropriate `import` statement in your program. For example, you have used the `Scanner` class, which is in the `java.util` package. This class requires the following `import` statement:

```
import java.util.Scanner;
```

You have also used various classes from the `java.io` package to perform file input and output. To import all of the classes from the `java.io` package, you use the following `import` statement:

```
import java.io.*;
```

The `java.lang` package is the only package that does not require an `import` statement. This package contains general classes, such as `String` and `System`, that are fundamental to the Java programming language. The `java.lang` package is automatically imported by all Java programs.

APPENDIX

H

# Working with Records and Random Access Files

## Structuring Data into Fields and Fixed-Length Records

Data that is written to a file is commonly structured as fields and records. In file terminology, a *field* is an individual piece of data, such as a person's name or telephone number. A *record* is a collection of fields pertaining to a single item. For example, a record might consist of a specific person's name, age, address, and telephone number.

Quite often you can save the contents of an object as a record in a file. You do this by writing each of the object's fields to the file, one after the other. When you have saved all of the object's fields, a complete record has been written. When the fields from multiple objects have been saved, then multiple records have been written to the file.

Random access files are particularly useful for storing and retrieving records. However, the sizes of the items stored in a random access file must be known in order to calculate the position of a specific item. Records that are stored in a random access file must be the same size and must have a *fixed length*. This means that the size of a record cannot change.

In Java, the sizes of the primitive data types are well documented and guaranteed to be the same on all systems. If an object's fields are all of the primitive data types, you can easily calculate the size of the record: it will be the sum of the sizes of all the fields. However, a problem arises if an object has a field that is a `String` because its contents can vary in length. You can get around this problem by making sure that a `String` field is always written as a specific number of characters. The example in this appendix shows one way to do this.

First we will introduce the `InventoryItem` class shown in Code Listing H-1. An object of this class can represent an item that a company might have in its inventory. This class has two fields: `description`, a `String` that holds an item's description, and `units`, an `int` that holds the number of units on hand. The class also has the necessary accessor methods, mutator methods, and two constructors.

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis. Copyright © 2016 by Pearson Education, Inc. All rights reserved.

**Code Listing H-1** (`InventoryItem.java`)

```
/**
  InventoryItem class
*/

public class InventoryItem
{
   private String description;  // Item description
   private int units;           // Units on hand

   /**
      This constructor assigns an empty string
      to description and 0 to units.
   */

   public InventoryItem()
   {
      description = "";
      units = 0;
   }

   /**
      This constructor assigns values
      to the description and units fields.
      @param d The description.
      @param u The units on hand.
   */

   public InventoryItem(String d, int u)
   {
      description = d;
      units = u;
   }

   /**
      The setDescription method assigns a string
      to the description field.
      @param d The string to assign to description.
   */

   public void setDescription(String d)
   {
      description = d;
   }

   /**
```

```
      The setUnits method assigns a value
      to the units field.
      @param u The value to assign to units.
   */

   public void setUnits(int u)
   {
      units = u;
   }

   /**
      The getDescription method returns the item's
      description.
      @return The description field.
   */

   public String getDescription()
   {
      return description;
   }

   /**
      The getUnits method returns the number of
      units on hand.
      @return The units field.
   */

   public int getUnits()
   {
      return units;
   }
}
```

The `InventoryItemFile` class shown in Code Listing H-2 is designed to read and write `InventoryItem` objects as records in a random access file. The class can also move the file pointer to a specific record. To keep the code simple, none of the exceptions are caught.

**Code Listing H-2** **(`InventoryItemFile.java`)**

```
import java.io.*;

/**
   This class manages a random access file which contains
   InventoryItem records.
*/
```

```

public class InventoryItemFile
{
   private final int RECORD_SIZE = 44;
   private RandomAccessFile inventoryFile;

   /**
      The constructor opens a random access file
      for both reading and writing.
      @param filename The name of the file.
      @exception  FileNotFoundException When the file
                  is not found.
 */

   public InventoryItemFile(String filename)
                         throws FileNotFoundException
   {
      // Open the file for reading and writing.
      inventoryFile =
           new RandomAccessFile(filename, "rw");
   }

   /**
      The writeInventoryItem method writes the contents
      of an InventoryItem object to the file at the
      current file pointer position.
      @param item The InventoryItem object to write.
      @exception IOException When a file error occurs.
   */

   public void writeInventoryItem(InventoryItem item)
                                  throws IOException
   {
      // Get the item's description.
      String str = item.getDescription();

      // Write the description.
      if (str.length() > 20)
      {
         // If there are more than 20 characters in the
         // string, then write only the first 20.
         for (int i = 0; i < 20; i++)
            inventoryFile.writeChar(str.charAt(i));
      }
      else
      {
         // Write the description to the file.
         inventoryFile.writeChars(str);
         // Write enough spaces to pad it out
         // to 20 characters.
         for (int i = 0; i < (20 - str.length()); i++)
            inventoryFile.writeChar(' ');
      }

      // Write the units to the file.
      inventoryFile.writeInt(item.getUnits());
   }

   /**
      The readInventoryItem method reads and returns
      the record at the current file pointer position.
      @return A reference to an InventoryItem object.
      @exception IOException When a file error occurs.
   */

   public InventoryItem readInventoryItem()
                              throws IOException
   {
      char[] charArray = new char[20];

      // Read the description, character by character,
      // from the file into the char array.
      for (int i = 0; i < 20; i++)
         charArray[i] = inventoryFile.readChar();

      // Store the char array in a String.
      String desc = new String(charArray);

      // Trim any trailing spaces from the string.
      desc.trim();

      // Read the units from the file.
      int u = inventoryFile.readInt();

      // Create an InventoryItem object and initialize
      // it with these values.
      InventoryItem item =
             new InventoryItem(desc, u);

      // Return the object.
      return item;
   }

   /**
      The getByteNum method returns a record's
      starting byte number.
      @param recordNum The record number of the
                       desired record.
   */

   private long getByteNum(long recordNum)
   {
      return RECORD_SIZE * recordNum;
   }

   /**
      The moveFilePointer method moves the file
      pointer to a specified record.
      @param recordNum  The number of the record to
                        move to.
      @exception IOException When a file error occurs.
   */

   public void moveFilePointer(long recordNum)
                          throws IOException
   {
      inventoryFile.seek(getByteNum(recordNum));
   }

   /**
      The getNumberOfRecords method returns the number
      of records stored in the file.
      @return The number of records in the file.
      @exception IOException When a file error occurs.
   */

   public long getNumberOfRecords() throws IOException
   {
      return inventoryFile.length() / RECORD_SIZE;
   }

   /**
      The close method closes the file.
      @exception IOException When a file error occurs.
   */

   public void close() throws IOException
   {
      inventoryFile.close();
   }
}
```

The `RECORD_SIZE` field, declared in line 10, is a `final int` variable initialized with the value 44. This is the size, in bytes, of a record. In a moment you will see how this number was determined. The `inventoryFile` field, declared in line 11, is a `RandomAccessFile` reference variable that will be used to open and work with a random access file. The constructor accepts a filename as a `String`. This filename is used to open a random access file, referenced by the `inventoryFile` variable, for reading and writing.

By looking at the `writeInventoryItem` method, in lines 37 through 63, we can see how the record size of 44 bytes was determined. The method accepts an `InventoryItem` object as an argument, the contents of which will be written as a record to the file. In line 41 the `description` field is retrieved and referenced by `str`, a local variable. Next, in lines 44 through 59, we write the `description` field to the file. To ensure that each record has the same fixed length, this method always writes the description as 20 characters. If the description has more than 20 characters, then only the first 20 are written. If the description has fewer than 20 characters, spaces are added to make up the difference. Next, in line 62, the method writes the `units` field, as an `int`, to the file.

Now we can see how the record size of 44 bytes was determined. When a character is written to the file, it is written as two bytes. The `description` field is written as 20 characters, so that's 40 bytes. The `units` field is written as an `int`, which uses 4 bytes. That makes a total record size of 44 bytes.

The `readInventoryItem` method in lines 72 through 98 reads a record from the file and returns an `InventoryItem` object containing the record's data. In line 75 the reference variable `charArray` is declared and a 20-element `char` array is created to hold the description. Then the code in lines 79 and 80 reads the 20 characters from the file and stores them in the array. Next, in line 83, a `String` object is created and the `char` array is passed as an argument. This copies the characters from the array to the `String` object.

If the description was less than 20 characters long, it will be padded with trailing spaces. The statement in line 86 trims any trailing spaces that might be in the string. Then the statement in line 89 reads the `units` field from the file and stores it in the `u` variable.

Now we can construct an `InventoryItem` object with the data we have read. This is done in lines 93 and 94. The last step, in line 97, is to return the object.

The class also has the ability to move the file pointer to a specific record. Two methods work together to perform this. First, `getByteNum` (in lines 107 through 110) is a private method that accepts a record number as an argument, and returns the record's starting byte number. It calculates the starting byte number by multiplying the record size by the record number. (The first record in the file is considered record 0.) The `moveFilePointer` method (in lines 120 through 124) accepts a record number as its argument, and moves the file pointer to the specified record. This method calls the `getByteNum` method to determine the record's starting location.

The `getNumberOfRecords` method appears in lines 133 through 136. This method returns the number of records in the file. It calculates the number of records by dividing the length of the file by the record size. The length of the file is returned by the `RandomAccessFile` class's `length` method.

The last method in the class is the `close` method, which closes the file. The program in Code Listing H-3 shows a simple demonstration of this class. This program asks the user to enter data for five items, which are stored in an array of `InventoryItem` objects. The program then saves the contents of the array elements to a file.

**Code Listing H-3** `(CreateInventoryFile.java)`

```java
import java.io.*;
import java.util.Scanner;

/**
   This program uses the InventoryFile class to create a
   file containing data from 5 InventoryItem objects.
*/

public class CreateInventoryFile
{
   public static void main(String[] args) throws IOException
   {
      final int NUM_ITEMS = 5;   // Number of items
      String description;        // Item description
      int units;                 // Units on hand

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create an array to hold InventoryItem objects.
      InventoryItem[] items = new InventoryItem[NUM_ITEMS];

      // Get data for the InventoryItem objects.
      System.out.println("Enter data for " + NUM_ITEMS +
                         " inventory items.");

      for (int i = 0; i < items.length; i++)
      {
         // Get the description.
         System.out.print("Enter an item description: ");
         description = keyboard.nextLine();

         // Get the units on hand.
         System.out.print("Enter the number of units: ");
         units = keyboard.nextInt();

         // Consume the remaining newline.
         keyboard.nextLine();

         // Create an InventoryItem object in the array.
         items[i] = new InventoryItem(description, units);
      }

      // Create an InventoryFile object.
      InventoryItemFile file =
                    new InventoryItemFile("Inventory.dat");

      // Write the contents of the array to the file.
      for (int i = 0; i < items.length; i++)
      {
         file.writeInventoryItem(items[i]);
      }

      // Close the file.
      file.close();

      System.out.println("The data was written to the " +
                         "Inventory.dat file.");
   }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter data for 5 inventory items.
Enter an item description: Wrench [Enter]
Enter the number of units: 20 [Enter]
Enter an item description: Hammer [Enter]
Enter the number of units: 15 [Enter]
Enter an item description: Pliers [Enter]
Enter the number of units: 12 [Enter]
Enter an item description: Screwdriver [Enter]
Enter the number of units: 25 [Enter]
Enter an item description: Ratchet [Enter]
Enter the number of units: 10 [Enter]
The data was written to the Inventory.dat file.
```

The program in Code Listing H-4 demonstrates how records can be randomly read from the file.

**Code Listing H-4** (`ReadInventoryFile.java`)

```
import java.io.*;
import java.util.Scanner;

/**
  This program displays specified records from
  the Inventory.dat file.
```

```
*/

public class ReadInventoryFile
{
   public static void main(String[] args) throws IOException
   {
      int recordNumber;          // Record number
      String again;              // To get a Y or an N
      InventoryItem item;        // An object from the file

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Open the file.
      InventoryItemFile file =
               new InventoryItemFile("Inventory.dat");

      // Report the number of records in the file.
      System.out.println("The Inventory.dat file has " +
               file.getNumberOfRecords() + " records.");

      // Get a record number from the user and
      // display the record.
      do
      {
         // Get the record number.
         System.out.print("Enter the number of the record " +
                          "you wish to see: ");
         recordNumber = keyboard.nextInt();

         // Consume the remaining newline.
         keyboard.nextLine();

         // Move the file pointer to that record.
         file.moveFilePointer(recordNumber);

         // Read the record at that location.
         item = file.readInventoryItem();

         // Display the record.
         System.out.println("\nDescription: " +
                            item.getDescription());
         System.out.println("Units: " + item.getUnits());

         // Ask the user whether to get another record.
         System.out.print("\nDo you want to see another " +
                          "record? (Y/N): ");
         again = keyboard.nextLine();
      } while (again.charAt(0) == 'Y' || again.charAt(0) == 'Y');

      // Close the file.
      file.close();
   }
}
```

**Program Output with Example Input Shown in Bold**

```
The Inventory.dat file has 5 records.
Enter the number of the record you wish to see: 4 [Enter]
Description: Ratchet
Units: 10
Do you want to see another record? (Y/N): y [Enter]
Enter the number of the record you wish to see: 2 [Enter]
Description: Pliers
Units: 12
Do you want to see another record? (Y/N): y [Enter]
Enter the number of the record you wish to see: 0 [Enter]
Description: Wrench
Units: 20
Do you want to see another record? (Y/N): y [Enter]
Enter the number of the record you wish to see: 1 [Enter]
Description: Hammer
Units: 15
Do you want to see another record? (Y/N): y [Enter]
Enter the number of the record you wish to see: 3 [Enter]
Description: Screwdriver
Units: 25
Do you want to see another record? (Y/N): n [Enter]
```

As a last demonstration, the program in Code Listing H-5 shows how an existing record in the file can be overwritten with a new record.

**Code Listing H-5** (`ModifyRecord.java`)

```
import java.io.*;
import java.util.Scanner;

/*
   This program allows the user to modify records in the
   Inventory.dat file.
*/

public class ModifyRecord
{
   public static void main(String[] args) throws IOException
   {
      int recordNumber;    // Record number
      int units;           // Units on hand
      String again;        // Want to change another one?
      String sure;         // Is the user sure?
      String description;  // Item description
      InventoryItem item;  // To reference an item

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Open the file.
      InventoryItemFile file =
              new InventoryItemFile("Inventory.dat");

      // Report the number of records in the file.
      System.out.println("The Inventory.dat file has " +
                         file.getNumberOfRecords() + " records.");

      // Get a record number from the user and
      // allow the user to modify it.
      do
      {
         // Get the record number.
         System.out.print("Enter the number of the record " +
                          "you wish to modify: ");
         recordNumber = keyboard.nextInt();

         // Consume the remaining newline.
         keyboard.nextLine();

         // Move the file pointer to that record number.
         file.moveFilePointer(recordNumber);

         // Read the record at that location.
         item = file.readInventoryItem();

         // Display the existing contents.
         System.out.println("Existing data:");
         System.out.println("\nDescription: " +
                            item.getDescription());
         System.out.println("Units: " + item.getUnits());

         // Get the new data.
         System.out.print("\nEnter the new description: ");
         description = keyboard.nextLine();
```

```
            System.out.print("Enter the number of units: ");
            units = keyboard.nextInt();
            keyboard.nextLine(); // Consume the remaining newline.

            // Store the new data in the object.
            item.setDescription(description);
            item.setUnits(units);

            // Make sure the user wants to save this data.
            System.out.print("Are you sure you want to save " +
                             "this data? (Y/N) ");
            sure = keyboard.nextLine();
            if (sure.charAt(0) == 'Y' || sure.charAt(0) == 'y')
            {
               // Move back to the record's starting position.
               file.moveFilePointer(recordNumber);
               // Save the new data.
               file.writeInventoryItem(item);
            }

            // Ask the user whether to change another record.
            System.out.print("\nDo you want to modify another " +
                             "record? (Y/N): ");
            again = keyboard.nextLine();
         } while (again.charAt(0) == 'Y' || again.charAt(0) == 'y');

         // Close the file.
         file.close();
      }
   }
```

### Program Output with Example Input Shown in Bold

```
The Inventory.dat file has 5 records.
Enter the number of the record you wish to modify: 3 [Enter]
Existing data:
Description: Screwdriver
Units: 25
Enter the new description: Duct Tape [Enter]
Enter the number of units: 30 [Enter]
Are you sure you want to save this data? (Y/N) y [Enter]
Do you want to modify another record? (Y/N): n [Enter]
```

In the example running of the program, record 3 was modified. We can run the `ReadInventoryFile` program in Code Listing H-4 again to verify that the record was changed. Here is the output of that program if we run it again.

**Program Output with Example Input Shown in Bold (`ReadInventoryFile.java`)**

```
The Inventory.dat file has 5 records.
Enter the number of the record you wish to see: 3 [Enter]
Description: Duct Tape
Units: 30
Do you want to see another record? (Y/N): n [Enter]
```

APPENDIX

# I Configuring Java DB and Installing Apache Derby

Beginning with Java 7, the Java DB database engine was bundled with the Java JDK. When you installed either Java 7 or Java 8, Java DB was automatically installed. Beginning with Java 9, however, Java DB is no longer bundled with the JDK. This is because Java DB is simply a rebranded version of Derby, an open source DBMS that is provided by the Apache Software Foundation. If you are using Java 9, you will need to separately download Derby from the Apache Software Foundation's website and install it. This appendix can help you configure Java DB, or install and configure Apache Derby, depending on the version of Java you are using.

## Configuring Java DB with Java 7 or Java 8

In Java 7 and Java 8, Java DB is automatically installed on your system when you install the JDK. When you install the JDK, take note of the location on your system where it is installed. If you accept the default location for the JDK, it will be installed in a folder with a path such as this:

```
C:\Program Files\Java\jdk1.8.0_101
```

The name of the last folder in the path might be different on your system, because it's name indicates the version of the JDK that you are using. Nevertheless, the path will be similar to the one shown here.

If you look inside the folder where the JDK is installed, you will see another folder named `db`. Inside the `db` folder, you will see another folder named `lib`, and inside the `lib` folder, you will see a file named *derby.jar*. The *derby.jar* file contains the Java DB database engine and its associated JBDC embedded driver. Take a moment to verify that you can locate the *derby.jar* file, and make a note of its path. For example, if you are using the JDK 8, update 101, the path of the *derby.jar* file will be something like:

```
C:\Program Files\Java\jdk1.8.0_101\db\lib\derby.jar
```

Once you have verified the path of the *derby.jar* file, you need to set the `CLASSPATH` environment variable on your system.

Access online material for this text at www.pearson.com/gaddis

## Setting the `CLASSPATH` Environment Variable

To use Java DB with this text, you will need to set your system's `CLASSPATH` environment variable to include the location of the *derby.jar* file. The `CLASSPATH` environment variable contains a list of class libraries that are needed by the Java Virtual Machine and other Java applications.

The following instructions describe the procedure for setting the `CLASSPATH` variable on Windows 10, Windows 8, and Windows 7.

### Windows 10

In the Windows search box, type "environment variables" then select **Edit the system environment variables** from the search results. This will display the *System Properties* window (with the *Advanced* tab selected at the top of the window). Click the Environment Variables . . . button. Look in the **System Variables** list to see if the `CLASSPATH` variable exists.

If the `CLASSPATH` variable already exists, perform the following:

- Select the `CLASSPATH` variable and click the **Edit** button. Add a semicolon to the end of the existing contents and then add the path of the `derby.jar` file to the end of the list.
- Click the OK buttons until all the dialog boxes are closed and exit the control panel.

If the `CLASSPATH` variable does not exist, perform the following:

- Click the New . . . button that appears below the list of system variables.
- In the Variable name box, type `CLASSPATH`.
- In the Variable value box, type a period, followed by a semicolon, followed by the path of the *derby.jar* file. For example, the value will be something similar to the following (depending on the version of the JDK that you are using):

  ```
  .;C:\Program Files\Java\jdk1.8.0_101\db\lib\derby.jar
  ```
- Click the OK buttons until all the dialog boxes are closed and exit the control panel.

### Windows 8

In the right bottom corner of the screen, click on the **Search** icon and type "Control Panel". Click on **Control Panel**, then click **System**, then click **Advanced system settings**. Click on the **Advanced** tab, then click **Environment Variables**. Look in the System Variables list to see if the `CLASSPATH` variable exists.

If the `CLASSPATH` variable already exists, perform the following:

- Select the `CLASSPATH` variable and click the **Edit** button. Add a semicolon to the end of the existing contents and then add the path of the *derby.jar* file to the end of the list.
- Click the OK buttons until all the dialog boxes are closed and exit the control panel.

If the `CLASSPATH` variable does not exist, perform the following:

- Click the New . . . button that appears below the list of system variables.
- In the Variable name box, type `CLASSPATH`.

- In the Variable value box, type a period, followed by a semicolon, followed by the path of the *derby.jar* file. For example, the value will be something similar to the following (depending on the version of the JDK that you are using):

```
.;C:\Program Files\Java\jdk1.8.0_101\db\lib\derby.jar
```

- Click the OK buttons until all the dialog boxes are closed and exit the control panel.

### Windows 7

Click the **Start button** and then right-click **Computer**. On the pop-up menu, select **Properties**. In the window that appears next, click **Advanced system settings**. This displays the System Properties window. Click the Environment Variables . . . button. Look in the System Variables list to see if the `CLASSPATH` variable exists.

If the `CLASSPATH` variable already exists, perform the following:

- Select the `CLASSPATH` variable and click the Edit button. Add a semicolon to the end of the existing contents and then add the path of the *derby.jar* file to the end of the list.
- Click the OK buttons until all the dialog boxes are closed and exit the control panel.

If the `CLASSPATH` variable does not exist, perform the following:

- Click the New . . . button that appears below the list of system variables.
- In the Variable name box, type `CLASSPATH`.
- In the Variable value box, type a period, followed by a semicolon, followed by the path of the *derby.jar* file. For example, the value will be something similar to the following (depending on the version of the JDK that you are using):

```
.;C:\Program Files\Java\jdk1.8.0_101\db\lib\derby.jar
```

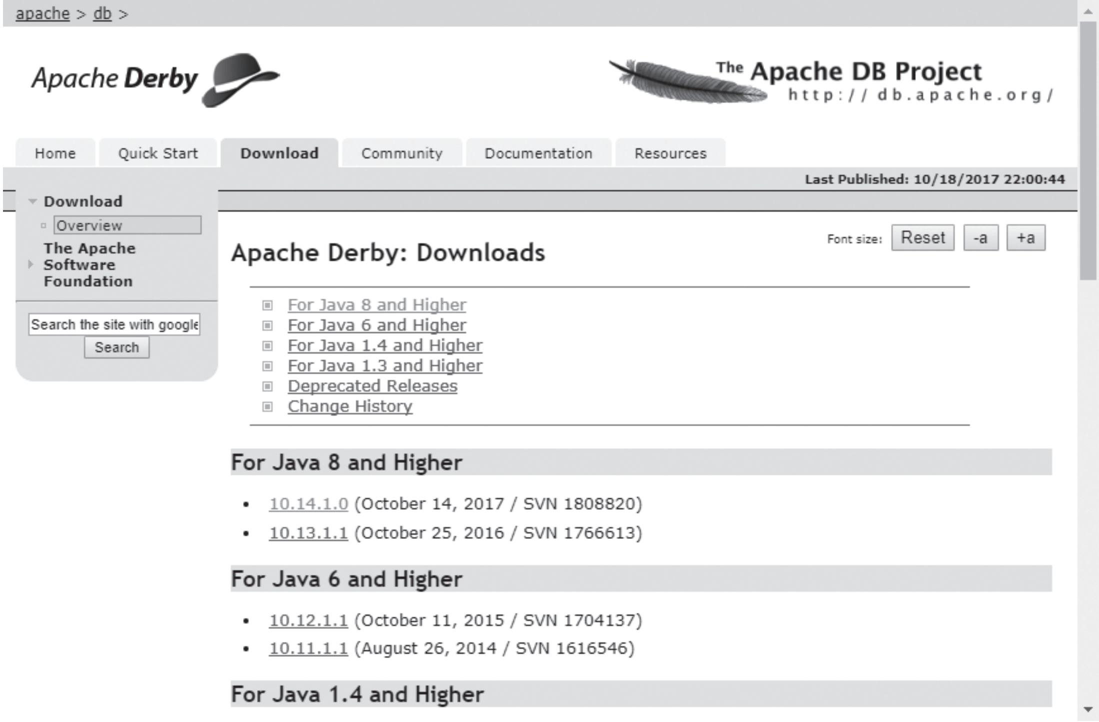
- Click the OK buttons until all the dialog boxes are closed and exit the control panel.

## Configuring Derby with Java 9

Derby is available as a free download from the Apache Software Foundation. To download Derby, follow these instructions:

1. Open your browser and go the following address: http://db.apache.org/derby/derby_downloads.html
2. The page that opens will look similar to Figure I-1. Look at the section titled *For Java 8 and Higher*, and click on the most recent version number that is listed. (The most recent version number will most likely be different than the one shown in the figure.)

**Figure I-1** The Apache Derby Download Page (Apache Software Foundation)


apache > db >
Apache Derby
The Apache DB Project
http://db.apache.org/
Home
Quick Start
Download
Community
Documentation
Resources
Last Published: 10/18/2017 22:00:44
Download
Overview
The Apache Software Foundation
Search the site with google
Search
Font size:
Reset
-a
+a

Apache Derby: Downloads

- For Java 8 and Higher
- For Java 6 and Higher
- For Java 1.4 and Higher
- For Java 1.3 and Higher
- Deprecated Releases
- Change History

For Java 8 and Higher

- 10.14.1.0 (October 14, 2017 / SVN 1808820)
- 10.13.1.1 (October 25, 2016 / SVN 1766613)

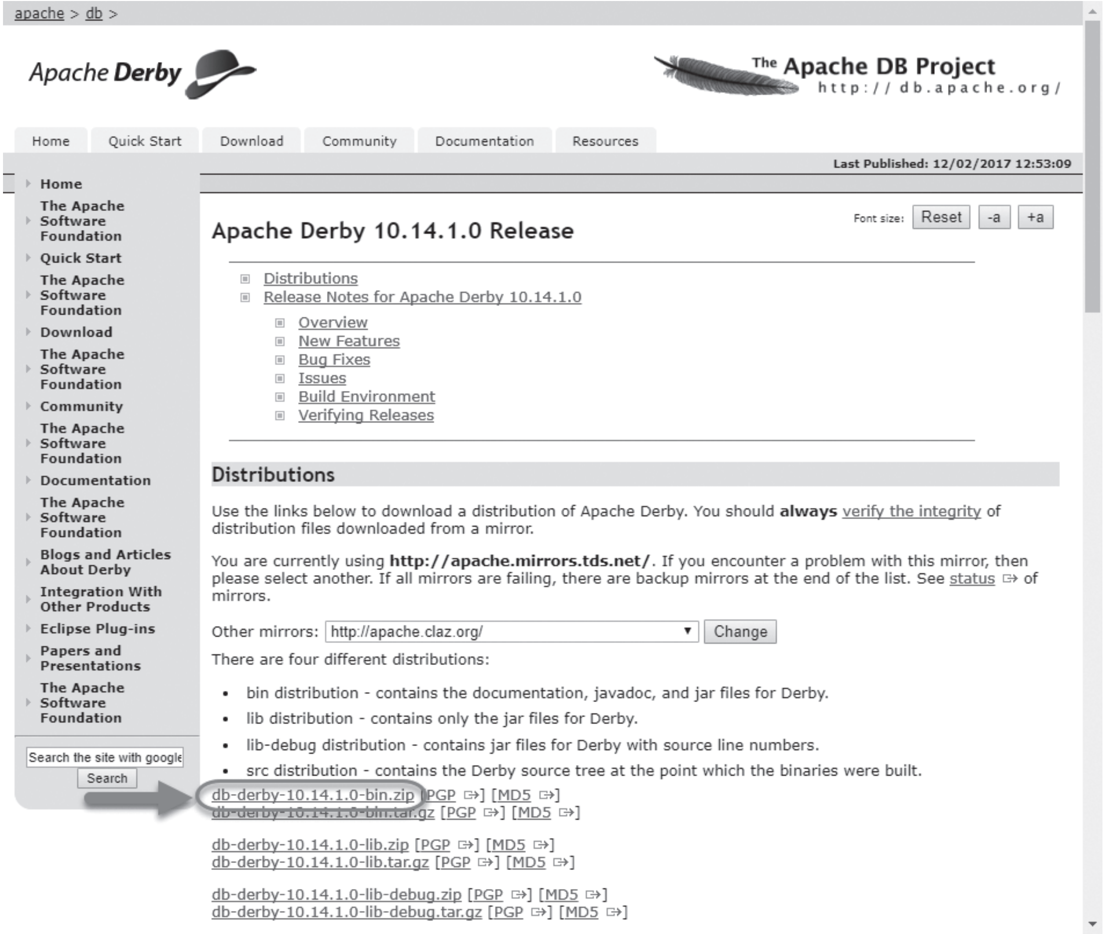
For Java 6 and Higher

- 10.12.1.1 (October 11, 2015 / SVN 1704137)
- 10.11.1.1 (August 26, 2014 / SVN 1616546)

For Java 1.4 and Higher

3. The page that opens will look similar to Figure I-2. In the *Distributions* section you will see links to download various different distributions. As shown in the figure, click on the link with the most recent version number ending with `bin.zip`. (The most recent version number will most likely be different than the one shown in the figure.)
4. The ZIP file that you downloaded contains a folder with a name such as `db-derby-10.14.1.0-bin`. (This name reflects the version number of the distribution. The name you see will be similar, but most likely it will be a different version number.) Extract this folder and its contents to a location on your system. Make a note of the location to which you extracted the folder. For example, on a Windows system, if you extracted the folder to the root folder of drive C: the path would be `C:\db-derby-10.14.1.0-bin`.

**Figure I-2** Select the latest `bin` distribution (Apache Software Foundation)


apache > db >
Apache Derby
The Apache DB Project
http://db.apache.org/
Home
Quick Start
Download
Community
Documentation
Resources
Home
The Apache Software Foundation
Quick Start
The Apache Software Foundation
Download
The Apache Software Foundation
Community
The Apache Software Foundation
Documentation
The Apache Software Foundation
Blogs and Articles About Derby
Integration With Other Products
Eclipse Plug-ins
Papers and Presentations
The Apache Software Foundation
Search the site with google
Search
Font size:
Reset
-a
+a
Change

Last Published: 12/02/2017 12:53:09

Apache Derby 10.14.1.0 Release

- Distributions
- Release Notes for Apache Derby 10.14.1.0
  - Overview
  - New Features
  - Bug Fixes
  - Issues
  - Build Environment
  - Verifying Releases

Distributions

Use the links below to download a distribution of Apache Derby. You should **always** verify the integrity of distribution files downloaded from a mirror.

You are currently using **http://apache.mirrors.tds.net/**. If you encounter a problem with this mirror, then please select another. If all mirrors are failing, there are backup mirrors at the end of the list. See status ⇨ of mirrors.

Other mirrors: http://apache.claz.org/

There are four different distributions:

- bin distribution - contains the documentation, javadoc, and jar files for Derby.
- lib distribution - contains only the jar files for Derby.
- lib-debug distribution - contains jar files for Derby with source line numbers.
- src distribution - contains the Derby source tree at the point which the binaries were built.

db-derby-10.14.1.0-bin.zip [PGP ⇨] [MD5 ⇨]
db-derby-10.14.1.0-bin.tar.gz [PGP ⇨] [MD5 ⇨]

db-derby-10.14.1.0-lib.zip [PGP ⇨] [MD5 ⇨]
db-derby-10.14.1.0-lib.tar.gz [PGP ⇨] [MD5 ⇨]

db-derby-10.14.1.0-lib-debug.zip [PGP ⇨] [MD5 ⇨]
db-derby-10.14.1.0-lib-debug.tar.gz [PGP ⇨] [MD5 ⇨]

## Setting the CLASSPATH Environment Variable

The Derby distribution file that you downloaded and extracted contains a file named `derby.jar`. This file is the Derby database engine. The `derby.jar` file will be located in the `lib` folder, which is in the distribution folder that you extracted. For example, suppose you extracted the `db-derby-10.14.1.0-bin` distribution file to the root folder of drive C:. The path of the `derby.jar` file would be:

```
C:\db-derby-10.14.1.0-bin\lib\derby.jar
```

You will need to configure your system so the Java Virtual Machine can find the `derby.jar` file. You do this by setting your system's `CLASSPATH` environment variable to include the path of the file. The following instructions describe the procedure for setting the `CLASSPATH` variable on Windows 10, Windows 8, and Windows 7.

### Windows 10

In the Windows search box, type *environment variables* and then select *Edit the system environment variables* from the search results. This will display the System Properties window (with the Advanced tab selected at the top of the window). Click the *Environment Variables . . .* button. Look in the System Variables list to see if the `CLASSPATH` variable exists.

If the `CLASSPATH` variable already exists, perform the following:

- Select the `CLASSPATH` variable and click the *Edit* button. Add a semicolon to the end of the existing contents and then add the path of the `derby.jar` file to the end of the list.
- Click the *OK* buttons until all the dialog boxes are closed and exit the control panel.

If the `CLASSPATH` variable does not exist, perform the following:

- Click the *New . . .* button that appears below the list of system variables.
- In the Variable name box, type `CLASSPATH`.
- In the Variable value box, type a period, followed by a semicolon, followed by the path of the `derby.jar` file. For example, the value will be something similar to the following (depending on the version of the Derby distribution that you are using):

```
.; C:\db-derby-10.14.1.0-bin\lib\derby.jar
```

- Click the *OK* buttons until all the dialog boxes are closed and exit the control panel.

### Windows 8

In the right bottom corner of the screen, click on the Search icon and type *Control Panel*. Click on *Control Panel*, then click *System*, then click *Advanced system settings*. Click on the *Advanced* tab, then click *Environment Variables*. Look in the System Variables list to see if the `CLASSPATH` variable exists.

If the `CLASSPATH` variable already exists, perform the following:

- Select the `CLASSPATH` variable and click the *Edit* button. Add a semicolon to the end of the existing contents and then add the path of the `derby.jar` file to the end of the list.
- Click the *OK* buttons until all the dialog boxes are closed and exit the control panel.

If the `CLASSPATH` variable does not exist, perform the following:

- Click the *New . . .* button that appears below the list of system variables.
- In the Variable name box, type `CLASSPATH`.
- In the Variable value box, type a period, followed by a semicolon, followed by the path of the `derby.jar` file. For example, the value will be something similar to the following (depending on the version of the Derby distribution that you are using):

```
.; C:\db-derby-10.14.1.0-bin\lib\derby.jar
```

- Click the *OK* buttons until all the dialog boxes are closed and exit the control panel.

### Windows 7

Click the Windows Start button and then right-click *Computer*. On the pop-up menu, select *Properties*. In the window that appears next, click *Advanced system settings*. This displays the System Properties window. Click the *Environment Variables* . . . button. Look in the System Variables list to see if the `CLASSPATH` variable exists.

If the `CLASSPATH` variable already exists, perform the following:

- Select the `CLASSPATH` variable and click the *Edit* button. Add a semicolon to the end of the existing contents and then add the path of the `derby.jar` file to the end of the list.
- Click the *OK* buttons until all the dialog boxes are closed and exit the control panel.

If the `CLASSPATH` variable does not exist, perform the following:

- Click the *New* . . . button that appears below the list of system variables.
- In the Variable name box, type `CLASSPATH`.
- In the Variable value box, type a period, followed by a semicolon, followed by the path of the `derby.jar` file. For example, the value will be something similar to the following (depending on the version of the Derby distribution that you are using):

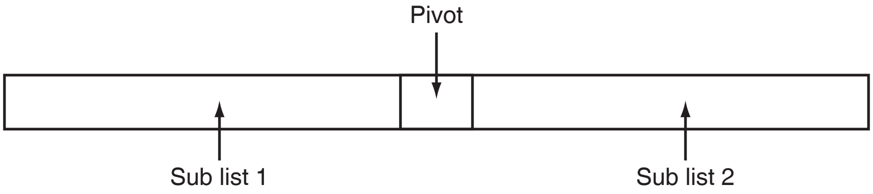
  ```
  .; C:\db-derby-10.14.1.0-bin\lib\derby.jar
  ```
- Click the *OK* buttons until all the dialog boxes are closed and exit the control panel.

APPENDIX J

# The QuickSort Algorithm

The QuickSort algorithm is a popular general-purpose sorting routine developed in 1960 by C. A. R. Hoare. It sorts an array by dividing it into two sub lists. Between the sub lists is a selected value known as the pivot. This is illustrated in Figure J-1.

**Figure J-1** Sub lists and pivot



Notice in the figure that sub list 1 is positioned to the left of (before) the pivot, and sub list 2 is positioned to the right of (after) the pivot. Once a pivot value has been selected, the algorithm swaps the other values in the array until all the elements in sub list 1 are less than the pivot, and all the elements in sub list 2 are greater than the pivot.

Once this is done, the algorithm repeats the procedure on sub list 1, then on sub list 2. The recursion stops when there is only one element in a sub list. At that point, the array is completely sorted.

The algorithm is coded primarily in two methods: `quickSort` and `partition`. The `quickSort` method is recursive. Its psuedocode is shown here:

```
quickSort:
    If Starting Index < Ending Index
        Partition the List around a Pivot
        quickSort Sub list 1
        quickSort Sub list 2
    End If
```

Access online material for this text at www.pearson.com/gaddis

Here is the Java code for the `quickSort` method:

```
public static void quickSort(int array[], int start, int end)
{
   int pivotPoint;

   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(array, start, end);

      // Sort the first sub list.
      quickSort(array, start, pivotPoint - 1);

      // Sort the second sub list.
      quickSort(array, pivotPoint + 1, end);
   }
}
```

This version of `quickSort` works with an array of integers. Its first argument is the array holding the list that is to be sorted. The second and third arguments are the starting and ending subscripts of the list.

The subscript of the pivot element is returned by the `partition` method. The `partition` method not only determines which element will be the pivot, but also controls the rearranging of the other values in the list. The method selects the element in the middle of the array as the pivot, and then scans the remainder of the array searching for values less than the pivot.

The code for the `partition` method is shown here:

```
private static int partition(int array[], int start, int end)
{
   int pivotValue, pivotIndex, mid;

   mid = (start + end) / 2;

   swap(array, start, mid);
   pivotIndex = start;
   pivotValue = array[start];
   for (int scan = start + 1; scan <= end; scan++)
   {
      if (array[scan] < pivotValue)
      {
         pivotIndex++;
         swap(array, pivotIndex, scan);
      }
   }
   swap(array, start, pivotIndex);

return pivotIndex;
}
```

The `partition` method does not initially sort the values into their final order. Its job is only to move the values that are less than the pivot to the pivot's left, and move the values that are greater than the pivot to the pivot's right. As long as that condition is met, they may appear in any order. The ultimate sorting order of the entire array is achieved cumulatively, though the recursive calls to the `quickSort` method.

There are many different ways of partitioning the array. As previously stated, the technique shown in this `partition` method selects the middle value as the pivot. That value is then moved to the beginning of the array (by swapping it with the value stored there). This simplifies the next step, which is to scan the array.

A `for` loop scans the remainder of the array, and when an element is found whose value is less than the pivot, that value is moved to a location left of the pivot point.

A third method, `swap`, is used to swap the values found in any two elements of the array. The method is shown here:

```
private static void swap(int[] array, int a, int b)
{
   int temp;

   temp = array[a];
   array[a] = array[b];
   array[b] = temp;
}
```

The program in Code Listing J-1 demonstrates these methods.

**Code Listing J-1** (`QuickSortDemo.java`)

```
/**
 * This program demonstrates the QuickSort algorithm.
 */

public class QuickSortDemo
{
   /**
    * main method
    */

   public static void main(String[] args)
   {
      int[] array = { 7, 3, 9, 2, 0, 1, 8, 4, 6, 5 };

      // Display the array as it is now.
      System.out.println("Before the sort:");
      for (int i = 0; i < 10; i++)
         System.out.print(array[i] + " ");
      System.out.println();

      // Sort the array.
      quickSort(array, 0, 9);

      // Display the array again.
      System.out.println("After the sort:");
      for (int i = 0; i < 10; i++)
         System.out.print(array[i] + " ");
      System.out.println();
   }

   /**
    *  The quickSort method uses the QuickSort algorithm to
    *  sort array, from array[start] through array[end].
    */

   public static void quickSort(int array[], int start, int end)
   {
      int pivotPoint;

      if (start < end)
      {
         // Get the pivot point.
         pivotPoint = partition(array, start, end);

         // Sort the first sub list.
         quickSort(array, start, pivotPoint - 1);

         // Sort the second sub list.
         quickSort(array, pivotPoint + 1, end);
      }
   }

   /**
    *  The partition method selects the value in the middle of
    *  the array as the pivot. The list is rearranged so all
    *  the values less than the pivot are on its left and all
    *  the values greater than pivot are on its right.
    */

   private static int partition(int array[], int start, int end)
   {
      int pivotValue, pivotIndex, mid;

      mid = (start + end) / 2;

      swap(array, start, mid);
      pivotIndex = start;
      pivotValue = array[start];
      for (int scan = start + 1; scan <= end; scan++)
      {
         if (array[scan] < pivotValue)
         {
            pivotIndex++;
            swap(array, pivotIndex, scan);
         }
      }
      swap(array, start, pivotIndex);

      return pivotIndex;
   }

   /**
    *  The swap method swaps the element at array[a] with the
    *  element at array[b].
    */

   private static void swap(int[] array, int a, int b)
   {
      int temp;

      temp = array[a];
      array[a] = array[b];
      array[b] = temp;
   }
}
```

**Program Output**

```
Before the sort:
7 3 9 2 0 1 8 4 6 5
After the sort:
0 1 2 3 4 5 6 7 8 9
```

# APPENDIX K Named Colors

aliceblue = #f0f8ff
antiquewhite = #faebd7
aqua = #00ffff
aquamarine = #7fffd4
azure = #f0ffff
beige = #f5f5dc
bisque = #ffe4c4
black = #000000
blanchedalmond = #ffebcd
blue = #0000ff
blueviolet = #8a2be2
brown = #a52a2a
burlywood = #deb887
cadetblue = #5f9ea0
chartreuse = #7fff00
chocolate = #d2691e
coral = #ff7f50
cornflowerblue = #6495ed
cornsilk = #fff8dc
crimson = #dc143c
cyan = #00ffff
darkblue = #00008b
darkcyan = #008b8b
darkgoldenrod = #b8860b
darkgray = #a9a9a9
darkgreen = #006400
darkgrey = #a9a9a9
darkkhaki = #bdb76b
darkmagenta = #8b008b
darkolivegreen = #556b2f
darkorange = #ff8c00
darkorchid = #9932cc
darkred = #8b0000
darksalmon = #e9967a
darkseagreen = #8fbc8f
darkslateblue = #483d8b
darkslategray = #2f4f4f
darkslategrey = #2f4f4f
darkturquoise = #00ced1
darkviolet = #9400d3
deeppink = #ff1493
deepskyblue = #00bfff
dimgray = #696969
dimgrey = #696969
dodgerblue = #1e90ff
firebrick = #b22222
floralwhite = #fffaf0
forestgreen = #228b22
fuchsia = #ff00ff
gainsboro = #dcdcdc
ghostwhite = #f8f8ff
gold = #ffd700
goldenrod = #daa520
gray = #808080
green = #008000
greenyellow = #adff2f
grey = #808080
honeydew = #f0fff0
hotpink = #ff69b4
indianred = #cd5c5c
indigo = #4b0082
ivory = #fffff0

Access online material for this text at www.pearson.com/gaddis

khaki = #f0e68c
lavender = #e6e6fa
lavenderblush = #fff0f5
lawngreen = #7cfc00
lemonchiffon = #fffacd
lightblue = #add8e6
lightcoral = #f08080
lightcyan = #e0ffff
lightgoldenrodyellow = #fafad2
lightgray = #d3d3d3
lightgreen = #90ee90
lightgrey = #d3d3d3
lightpink = #ffb6c1
lightsalmon = #ffa07a
lightseagreen = #20b2aa
lightskyblue = #87cefa
lightslategray = #778899
lightslategrey = #778899
lightsteelblue = #b0c4de
lightyellow = #ffffe0
lime = #00ff00
limegreen = #32cd32
linen = #faf0e6
magenta = #ff00ff
maroon = #800000
mediumaquamarine = #66cdaa
mediumblue = #0000cd
mediumorchid = #ba55d3
mediumpurple = #9370db
mediumseagreen = #3cb371
mediumslateblue = #7b68ee
mediumspringgreen = #00fa9a
mediumturquoise = #48d1cc
mediumvioletred = #c71585
midnightblue = #191970
mintcream = #f5fffa
mistyrose = #ffe4e1
moccasin = #ffe4b5
navajowhite = #ffdead
navy = #000080
oldlace = #fdf5e6
olive = #808000
olivedrab = #6b8e23
orange = #ffa500
orangered = #ff4500
orchid = #da70d6
palegoldenrod = #eee8aa
palegreen = #98fb98
paleturquoise = #afeeee
palevioletred = #db7093
papayawhip = #ffefd5
peachpuff = #ffdab9
peru = #cd853f
pink = #ffc0cb
plum = #dda0dd
powderblue = #b0e0e6
purple = #800080
red = #ff0000
rosybrown = #bc8f8f
royalblue = #4169e1
saddlebrown = #8b4513
salmon = #fa8072
sandybrown = #f4a460
seagreen = #2e8b57
seashell = #fff5ee
sienna = #a0522d
silver = #c0c0c0
skyblue = #87ceeb
slateblue = #6a5acd
slategray = #708090
slategrey = #708090
snow = #fffafa
springgreen = #00ff7f
steelblue = #4682b4
tan = #d2b48c
teal = #008080
thistle = #d8bfd8
tomato = #ff6347
transparent = rgba(0,0,0,0)
turquoise = #40e0d0
violet = #ee82ee
wheat = #f5deb3
white = #ffffff
whitesmoke = #f5f5f5
yellow = #ffff00
yellowgreen = #9acd32

# APPENDIX L

# Answers to Checkpoints

## Chapter 1

1.1 Because the computer can be programmed to do so many different tasks.

1.2 The Central Processing Unit (CPU), main memory, secondary storage devices, input devices, output devices.

1.3 Arithmetic and Logic Unit (ALU), and Control Unit

1.4 Fetch: The CPU's control unit fetches the program's next instruction from main memory.

Decode: The control unit decodes the instruction, which is encoded in the form of a number. An electrical signal is generated.

Execute: The signal is routed to the appropriate component of the computer, which causes a device to perform an operation.

1.5 A unique number assigned to each section of memory. The address is used to identify a location in memory.

1.6 Program instructions and data are stored in main memory while the program is operating. Main memory is volatile, and loses its contents when power is removed from the computer. Secondary storage holds data for long periods of time—even when there is no power to the computer.

1.7 It means that an operating system is capable of running multiple programs at once.

1.8 A key word has a special purpose, and is defined as part of a programming language. A programmer-defined symbol is a word or name defined by the programmer.

1.9 Operators perform operations on one or more operands. Punctuation symbols mark the beginning or ending of a statement, or separates items in a list.

1.10 A line is a single line as it appears in the body of a program. A statement is a complete instruction that causes the computer to perform an action.

1.11 Because their contents may be changed.

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Data Structures*, Fourth Edition.
Tony Gaddis. Copyright © 2019 by Pearson Education, Inc. All rights reserved.

1.12 The original value is overwritten.

1.13 A compiler is a program that translates source code into an executable form.

1.14 Syntax errors are mistakes that the programmer has made that violate the rules of the programming language. These errors must be corrected before the compiler can translate the source code.

1.15 The Java compiler translates Java source code into byte code, which is an intermediate language. The Java Virtual Machine executes the byte code instructions.

1.16 The Java Virtual Machine (JVM) is a program that reads Java byte code instructions and executes them as they are read. In other words, it interprets byte code instructions. You can think of the JVM as a program that simulates a computer whose machine language is Java byte code.

1.17 The program's purpose, input, process, and output.

1.18 Before you create a program on the computer, it is often a good idea to imagine what the computer screen will look like while the program is running. If it helps, draw pictures of the screen, with sample input and output, at various points in the program.

1.19 A cross between human language and a programming language. Pseudocode is especially helpful when designing an algorithm. Although the computer can't understand pseudocode, programmers often find it helpful to write an algorithm in a language that's "almost" a programming language, but still very similar to natural language.

1.20 A compiler translates source code into an executable form.

1.21 A runtime error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

1.22 Syntax errors are found by the compiler.

1.23 You can provide sample data and predict what the output should be. If the program does not produce the correct output, a logical error is present in the program.

1.24 Data and the code that operates on the data.

1.25 The data contained in an object.

1.26 The procedures, or behaviors, that an object performs.

1.27 Encapsulation refers to the combining of data and code into a single object.

1.28 Data hiding refers to an object's ability to hide its data from code that is outside the object. Only the object's methods may then directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access the methods that operate on the data.

# Chapter 2

2.1
```
// A crazy mixed up program
public class Columbus
{
   public static void main(String[] args)
   {
      System.out.println("In 1492 Columbus sailed the ocean blue.");
   }
}
```

2.2 `Columbus.java`

2.3
```
public class Hello
{
   public static void main(String[] args)
   {
      System.out.println("Hello World");
   }
}
```

2.4
```
// Example
// August 22, 2018
public class MyName
{
   public static void main(String[] args)
   {
      System.out.println("Herbert Dorfmann");
   }
}
```

2.5 C

2.6 A

2.7
```
// Its a mad, mad program
public class Success
{
   public static void main(String[] args)
   {
      System.out.print("Success\n");
      System.out.print("Success ");
      System.out.print("Success\n");
      System.out.println("\nSuccess");
   }
}
```

2.8
```
The works of Wolfgang
include the following
The Turkish March and Symphony No. 40 in G minor.
```

2.9
```
// August 22, 2018
public class PersonalInfo
{
   public static void main(String[] args)
   {
      System.out.println("Herbert Dorfmann");
      System.out.println("123 Elm Street");
      System.out.println("My Town, NC 21111");
      System.out.println("919-555-1234");
   }
}
```

2.10 *Variables:*
`little`
`big`
*Literals:*
`2`
`2000`
`"The little number is "`
`"The big number is "`

2.11 `The value is number`

2.12 `99bottles` is illegal because it starts with a number.
`r&d` is illegal because the `&` character is illegal.

2.13 They are not the same because one begins with an uppercase S while the other begins with a lowercase s. Variable names are case-sensitive.

2.14
a. `short`
b. `int`
c. 22.1 because it is stored as a `double`.

2.15 6.31E17

2.16 Append the `F` suffix to the numeric literal, such as:

```
number = 7.4F;
```

2.17 `true` and `false`

2.18
a. `char letter;`
b. `letter = 'A';`
c. `System.out.println(letter);`

2.19 The code for ‘C’ is 67.
The code for ‘F’ is 70.
The code for ‘W’ is 87.

2.20 `'B'` is a character literal.

2.21 You cannot assign a string literal to a `char` variable. The statement should be:

```
char letter = 'Z';
```

2.22

| **Expression** | **Value** |
|---|---|
| `6 + 3 * 5` | `21` |
| `12 / 2 – 4` | `2` |
| `9 + 14 * 2 – 6` | `31` |
| `5 + 19 % 3 – 1` | `5` |
| `(6 + 2) * 3` | `24` |
| `14 / (11 – 4)` | `2` |
| `9 + 12 * (8 – 3)` | `69` |

2.23 Integer division. The value 23.0 will be stored in `portion`.

2.24
a. `x += 6;`
b. `amount –= 4;`
c. `y *= 4;`
d. `total /= 27;`
e. `x %= 7;`

2.25
a. No
b. Because the result of `basePay + bonus` results in an `int` value, which cannot be stored in the `short` variable `totalPay`. You can fix the problem by declaring `totalPay` as an `int`, or casting the result of the expression to a `short`.

2.26 `a = (float)b;`

2.27 `String city = "San Francisco";`

2.28 `stringLength = city.length();`

2.29 `oneChar = city.charAt(0);`

2.30 `upperCity = city.toUpperCase();`

2.31 `lowerCity = city.toLowerCase();`

2.32 To write a single line comment you begin the comment with `//`. To write a multi-line comment you begin the comment with `/*` and end it with `*/`. To write a documentation comment you begin the comment with `/**` and end it with `*/`.

2.33 Documentation comments can be read and processed by a program named `javadoc`. The `javadoc` program can read Java source code files and generate attractively formatted HTML documentation files. If the source code files contain any documentation comments, the information in the comments becomes part of the HTML documentation.

2.34 A message dialog box is used to display a message to the user. An input dialog box is used to gather keyboard input from the user.

2.35
a. `JOptionPane.showMessageDialog(null, "Greetings Earthling.");`
b. `str = JOptionPane.showInputDialog("Enter a number.");`

2.36

```
String str;
int age;
str = JOptionPane.showInputDialog("Enter your age.");
age = Integer.parseInt(str);
```

2.37 `import javax.swing.JOptionPane;`

## Chapter 3

3.1
```
if (y == 20)
   x = 0;
```

3.2
```
if (hours > 40)
   payRate *= 1.5;
```

3.3
```
if (sales >= 10000)
   commission = 0.2;
```

3.4
```
if (max)
   fees = 50;
```

3.5
```
if (x > 100)
{
   y = 20;
   z = 40;
}
```

3.6
```
if (a < 10)
{
   b = 0;
   c = 1;
}
```

3.7
```
if (myCharacter == 'D')
   System.out.println("Goodbye");
```

3.8
```
if (x > 100)
   y = 20;
else
   y = 0;
```

3.9
```
if (y == 100)
   x = 1;
else
   x = 0;
```

3.10
```
if (sales >= 50000.0)
   commission = 0.2;
else
   commission = 0.1;
```

3.11
```
if (a < 10)
{
   b = 0;
   c = 1;
}
else
{
   b = -99;
   c = 0;
}
```

3.12 `1 1`

3.13

| If the customer purchases this many books . . . | this many coupons are given. |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 10 | 3 |

3.14

```
if (amount1 > 10)
   {
     if (amount2 < 100)
     {
        if (amount1 > amount2)
           System.out.println(amount1);
         else
           System.out.println(amount2);
     }
   }
```

3.15

```
if (x > 0)
{
   if (y < 20)
   {
      z = 1;
   }
   else
   {
      z = 0;
   }
}
```

3.16

| **Logical Expression** | **Result (true or false)** |
|---|---|
| `true && false` | `false` |
| `true && true` | `true` |
| `false && true` | `false` |
| `false && false` | `false` |
| `true \|\| false` | `true` |
| `true \|\| true` | `true` |
| `false \|\| true` | `true` |
| `false \|\| false` | `false` |
| `!true` | `false` |
| `!false` | `true` |

3.17 T, F, T, T, T

3.18

```
if (speed >= 0 && speed <= 200)
   System.out.println("The number is valid");
```

3.19
```
if (speed < 0 || speed > 200)
   System.out.println("The number is not valid");
```

3.20
```
if (name.equals("Timothy"))
   System.out.println("Do I know you?");
```

3.21
```
if (name1.compareTo(name2) < 0)
   System.out.println(name1 + " " + name2);
else
   System.out.println(name2 + " " + name1);
```

3.22
```
if (name.equalsIgnoreCase("Timothy"))
   System.out.println("Do I know you?");
```

3.23

a. `z = x > y ? 1 : 20;`

b. `population = temp > 45 ? base * 10 : base * 2;`

c. `wages = hours > 40 ? wages * 1.5 : wages * 1;`

d.
```
System.out.println(result >=0 ? "The result is positive" :
                                "The result is negative");
```

3.24
```
// Here is the switch statement.
switch(userNum)
{
   case 1 : System.out.println("One");
            break;
   case 2 : System.out.println("Two");
            break;
   case 3 : System.out.println("Three");
            break;
   default: System.out.println("Error: invalid number.");
}
```

3.25
```
switch(selection)
{
   case 'A' : System.out.println("You selected A.");
              break;
   case 'B' : System.out.println("You selected B.");
              break;
   case 'C' : System.out.println("You selected C.");
              break;
   case 'D' : System.out.println("You selected D.");
              break;
   default : System.out.println("Not good with letters, eh?");
}
```

3.26 Because it uses greater-than and less-than operators in the comparisons.

3.27 The case expressions must be a literal or a `final` variable, which must be of the `char`, `byte`, `short`, `int`, or `String` types. In this code, relational expressions are used.

3.28
```
That is serious.
```

3.29 `System.out.printf("%,.2f", number);`

3.30 `System.out.printf("%10.1f", number);`

3.31 `System.out.printf("%08.1f", number);`

3.32 `System.out.printf("%,10d", number);`

3.33 `System.out.printf("%-,20.2f", number);`

3.34 `System.out.printf("%20s", name);`

## Chapter 4

4.1 a. 2
b. 2
c. 1
d. 8

4.2 0 times

4.3 This must be a trick question. The statement that prints “I love Java programming!” is not in the body of the loop. The only statement that is in the body of the loop is the one that prints “Hello World”. Because there is no code in the loop to change the contents of the `count` variable, the loop will execute infinitely, printing “Hello World”. So, “I love Java programming!” is never printed.

4.4
```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number in the range of 10 - 24: ");
number = keyboard.nextInt();
while (number < 10 || number > 24)
{
    System.out.println("That number is not in the range.");
    System.out.print("Enter a number in the range of 10 – 24: ");
    number = keyboard.nextInt();
}
```

4.5
```
String input;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter Y, y, N, or n: ");
input = keyboard.nextLine();
ch = input.charAt(0);
while (ch != 'Y' && ch != 'y' && ch != 'N' && ch != 'n')
{
    System.out.println("Try again.");
    System.out.print("Enter Y, y, N, or n: ");
    input = keyboard.nextLine();
    ch = input.charAt(0);
}
```

4.6

```
String str;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter Yes or No: ");
str = keyboard.nextLine();
while ((!str.equals("Yes")) && (!str.equals("No")))
{
    System.out.print("Please enter Yes or No: ");
    str = keyboard.nextLine();
}
```

4.7 Initialization, test, and update.

4.8
a. `count = 1`
b. `count <= 50`
c. `count++`
d.

```
for (int count = 1; count <= 50; count++)
   System.out.println("I love to program");
```

4.9
a.

```
 0
 2
 4
 6
 8
10
```

b.

```
-5
-4
-3
-2
-1
 0
 1
 2
 3
 4
```

c.

```
 5
 8
11
14
17
```

4.10

```
// This is how Chloe Ashlyn would write the loop.
for (int i = 1; i <= 10; i++)
   System.out.println("Chloe Ashlyn");
```

4.11

```
for (int i = 1; i <= 49; i += 2)
   System.out.println(i);
```

4.12

```
for (int i = 0; i <= 100; i += 5)
   System.out.println(i);
```

4.13
```
Scanner keyboard = new Scanner(System.in);
int number = 0, total = 0;
for (int i = 1; i <= 7; i++)
{
   System.out.print("Enter a number: ");
   number = keyboard.nextInt();
   total += number;
}
```

4.14 The variable `x` is the loop control variable and `y` is the accumulator.

4.15 You should be careful to choose a value that cannot be mistaken as a valid input value.

4.16 Data is read from an input file, and data is written to an output file.

4.17 `import java.io.*;`

4.18 `PrintWriter`

4.19
```
// This is how Kathryn would write the code.
PrintWriter outputFile = new PrintWriter("MyName.txt");
outputFile.println("Kathryn");
outputFile.close();
```

4.20 `File` and `Scanner`

4.21
```
File file = new File("MyName.txt");
Scanner inputFile = new Scanner(file);
if (inputFile.hasNext())
{
   String str = inputFile.nextLine();
   System.out.println(str);
}
inputFile.close();
```

4.22 You create an instance of the `FileWriter` class to open the file. You pass the name of the file (a string) as the constructor's first argument, and the `boolean` value `true` as the second argument. Then, when you create an instance of the `PrintWriter` class, you pass a reference to the `FileWriter` object as an argument to the `PrintWriter` constructor. The file will not be erased if it already exists and new data will be written to the end of the file.

4.23 `throws IOException`

4.24 It assigns the variable `x` a random integer in the range −2,147,483,648 to +2,147,483,647.

4.25 It assigns the variable `x` a random integer in the range 0 through 99.

4.26 It assigns the variable `x` a random integer in the range 1 through 9.

4.27 It assigns the variable `x` a random number in the range 0.0 to 1.0.

## Chapter 5

5.1 A value-returning method returns a value back to the code that called it, and a `void` method does not.

5.2 Method call

5.3 Method header

5.4 If the user enters 5 the program will display *Able was I*. If the user enters 10 the program will display *I saw Elba*. If the user enters 100 the program will display *I saw Elba*.

5.5
```
// This is how Mary Catherine Jones would write it.
public static void myName()
{
   System.out.println("Mary Catherine Jones");
}
```

5.6 An argument is a value that is passed into a method. A parameter is a special variable that holds the value being passed into the method.

5.7 b and c will cause a compiler error because the values being sent as arguments (a `double` and a `long`) cannot be automatically converted to an `int`.

5.8 Only d is written correctly.

5.9 Only a copy of an argument's value is passed into a parameter variable. A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If a parameter variable is changed inside a method, it has no effect on the original argument.

5.10
```
99 1.5
99 1.5
0 0.0
99 1.5
```

5.11 `double`

5.12 `public static int days(int years, int months, int weeks)`

5.13 `public static double distance(double rate, double time)`

5.14 `public static long lightYears(long miles)`

## Chapter 6

6.1 To store data

6.2 Operations that the object can perform

6.3 It serves a purpose similar to that of the blueprint for a house. The blueprint itself is not a house, but rather a detailed description of a house. When we use the blueprint to build an actual house, we could say that we are building an instance of the house described by the blueprint. A class is not an object, but a description of an object. When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an instance of the class.

6.4 In the Java API

6.5 The `new` operator creates an object in memory, and returns that object's memory address.

6.6 It holds an object's memory address.

6.7 In order to fly a kite, you need a spool of string attached to it. When the kite is airborne, you use the spool of string to hold on to the kite and control it. This is similar to the relationship between an object and the variable that references the object. Think of the object as the kite, and the reference variable as the spool of string.

6.8 Classes are the blueprints.

6.9 The kite represents an object.

6.10 The memory address of the object.

6.11 A `String` object.

6.12 a. `Car`
  b. `make` and `yearModel`
  c. `setMake`, `setYearModel`, `getMake`, and `getYearModel`
  d. `make` and `yearModel`
  e. `setMake`, `setYearModel`, `getMake`, and `getYearModel`

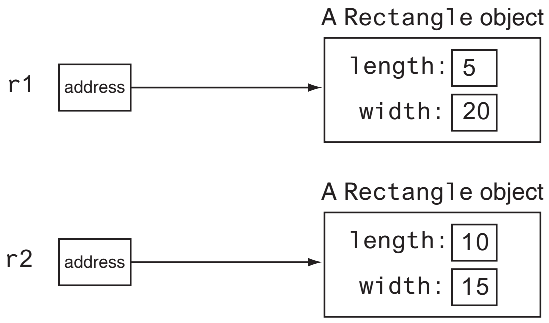
6.13 `limo.setMake("Cadillac");`

6.14 Creates an instance of an object in memory.

6.15 An accessor is a method that gets a value from a class's field but does not change it. A mutator is a method that stores a value in a field or in some other way changes the value of a field.

6.16 When the value of an item is dependent on other data and that item is not updated when the other data is changed, it is said that the item has become stale.

6.17



6.18 It has the same name as the class.

6.19 It has no return type, not even `void`.

6.20 a. `ClassAct`
b. `ClassAct myact = new ClassAct(25);`

6.21 Overloaded methods must have different parameter lists. Their return types do not matter.

6.22 A method's signature consists of the method's name and the data types of the method's parameters, in the order that they appear.

6.23
```
This is the second version of the method.
This is the first version of the method.
```

6.24 Only one.

6.25 The problem domain is the set of real-world objects, parties, and major events related to a problem.

6.26 Someone who has an adequate understanding of the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

6.27 Identify all the nouns (including pronouns and noun phrases) in the problem domain description. Each of these is a potential class. Then, refine the list to include only the classes that are relevant to the problem.

6.28 A class's responsibilities are the things that the class is responsible for knowing and the actions that the class is responsible for doing.

6.29 It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?"

6.30 No. Often responsibilities are discovered through brainstorming.

## Chapter 7

7.1 a. `int[] employeeNumbers = new int[100];`
b. `double[] payRates = new double[25];`
c. `float[] miles = new float[14];`
d. `char[] letters = new char[1000];`

7.2 An array's size declarator must be a non-negative integer expression. The first statement is incorrect because the size declarator is negative. The second statement is incorrect because the size declarator is a floating-point number.

7.3 0 through 3

7.4 The size declarator specifies the number of elements in the array. A subscript identifies a specific element in the array.

7.5 The subscript is outside the range of valid subscripts for the array.

7.6 When the statement executes, it crashes the program and displays a runtime error message.

7.7

```
1
2
3
4
5
```

7.8

```
double[] array = { 1.7, 6.4, 8.9, 3.1, 9.2 };
```

There are five elements in the array.

7.9

```
result = numbers1[0] * numbers2[3];
```

7.10

```
for (int i = 0; i < array.length; i++)
   array[i] = -1;
```

7.11

```
// Assume keyboard references a Scanner object.
int size;
System.out.print("Enter the size of the array: ");
size = keyboard.nextInt();
values = new double[size];
```

7.12

```
for (int i = 0; i < a.length; i++)
   b[i] = a[i];
```

7.13

```
myMethod(numbers);
```

7.14

```
public static void zero(int[] array)
{
   for (int i = 0; i < array.length; i++)
      array[i] = 0;
}
```

7.15

a.

```
String[] planets = { "Mercury", "Venus", "Earth", "Mars" };
```

b.

```
for (int i = 0; i < planets.length; i++)
   System.out.println(planets[i]);
```

c.

```
for (int i = 0; i < planets.length; i++)
   System.out.println(planets[i].charAt(0));
```

7.16

```
Rectangle[] rectArray = new Rectangle[5];
for (int i = 0; i <= rectArray.length; i++)
{
   // Initialize each rectangle with the values
   // i and i+1 for length and width.
   rectArray[i] = new Rectangle(i, i+1);
}
```

7.17

```
final int RACKS = 50;
final int SHELVES = 10;
final int VIDEOS = 25;
// Create an array to hold video numbers.
int videoNumbers[][][] = new int[RACKS][SHELVES][VIDEOS];
```

7.18 The selection sort first looks for the smallest value in the array. When it finds it, it moves it to element 0.

7.19 Only once.

7.20 The sequential search steps through each element of the array, starting at element 0, looking for the search value. The binary search requires that the array be sorted in ascending order. It starts by looking at the middle element. If it is not the search value, and is greater than the search value, then the lower half of the array is searched next. If the middle element is not the search value, and is less than the search value, the upper half of the array is searched next. This same technique is repeated on the half of the array being searched until the element is either found or there are no more elements to search.

7.21 10,000 comparisons

7.22 Move the items that are frequently searched for to the beginning of the array.

7.23 `import java.util.ArrayList;`

7.24 `ArrayList frogs = new ArrayList();`

7.25 `ArrayList<String> lizards = new ArrayList<String>();`

7.26 You use the `add` method.

7.27 The `ArrayList` class has a `remove` method that removes an item at a specific index. You pass the index as an argument to the method.

7.28 The `ArrayList` class has a `get` method that retrieves an item at a specific index. You pass the index as an argument to the method.

7.29 Inserting means adding an item at a specific index. The `ArrayList` class has an overloaded version of the `add` method that allows you to add an item at a specific index.

7.30 The `ArrayList` class has a `size` method that returns the number of items stored in the `ArrayList`.

7.31 An `ArrayList`'s size is the number of items stored in the `ArrayList` object. An `ArrayList`'s capacity is the number of items the `ArrayList` object can hold without having to increase its size.

## Chapter 8

8.1 Each instance of a class has its own copy of the class's instance fields. A static field does not belong to any instance of the class, and there is only one copy of a static field in memory, regardless of the number of instances of the class.

8.2 It isn't necessary for an instance of the class to be created in order to execute the method.

8.3 They cannot directly refer to non-static members of the class. This means that any method called from a static method must also be static. It also means that if the method uses any of the class's fields, they must be static as well.

8.4 The `this` variable will reference the `stock2` object.

8.5
- a. `Flower`
- b. The ordinal value for `Rose` is 0. The ordinal value for `DAISY` is 1. The ordinal value for `PETUNIA` is 2.
- c. `Flower.Rose`, `Flower.DAISY`, `Flower.PETUNIA`
- d. `Flower flora = Flower.PETUNIA;`

8.6

```
HOBBIT ELF DRAGON
```

8.7

```
Z is not greater than X.
```

## Chapter 9

9.1

```
little = Character.toLowerCase(big);
```

9.2

```
if (Character.isDigit(ch))
   System.out.println("digit");
else
   System.out.println("Not a digit");
```

9.3

```
A
```

9.4

```
String input;
char ch;
Scanner keyboard = new Scanner(System.in);
System.out.print("Do you want to repeat the " +
                 "program or quit? (R/Q)");
input = keyboard.nextLine();
ch = input.charAt(0);
ch = Character.toUpperCase(ch);
while (ch != 'R' && ch != 'Q')
{
   System.out.print("Do you want to repeat the " +
                    "program or quit? (R/Q)");
   input = keyboard.nextLine();
   ch = input.charAt(0);
   ch = Character.toUpperCase(ch);
}
```

9.5

```
$
```

9.6

```
int total = 0;
for (int i = 0; i < str.length(); i++)
{
   if (Character.isUpperCase(str.charAt(i)))
      total++;
}
```

9.7

```
public static boolean endsWithGer(String str)
{
   boolean status;

   if (str.endsWith("ger"))
      status = true;
   else
      status = false;
   return status;
}
```

9.8

```
public static boolean endsWithGer(String str)
{
   boolean status;
   String strUpper = str.toUpperCase();

   if (strUpper.endsWith("GER"))
      status = true;
   else
      status = false;
   return status;
}
```

9.9 You would use the `substring` method.

9.10 The `indexOf` method searches for a character or substring, starting at the beginning of a string. The `lastIndexOf` method searches for a character or substring, starting at the end of a string. This means that the `indexOf` method returns the index of the first occurrence of a character or substring, and the `lastIndexOf` method returns the index of the last occurrence of a character or substring.

9.11 The `substring` method returns a reference to a substring. The `getChars` method stores a substring in a `char` array.

9.12 The `concat` method.

9.13 The `toCharArray` method returns all of the characters in the calling object as a `char` array.

9.14 The fellow student is wrong. The `replace` method will return a reference to a `String` which is a copy of `str1`, in which all of the `o` characters have been replaced with `u` characters. The original string in `str1` will not be changed, however. The code will produce the following output:

```
To be, or not to be
Tu be, ur nut tu be
```

9.15

```
WilliamtheConqueror
```

9.16

```
str = String.valueOf(number);
```

9.17 Once you create a `String` object, you cannot change the object's value.

9.18 It would be more efficient to use `StringBuilder` objects because they are not immutable. Making a change to a `StringBuilder` object does not cause a new object to be created.

9.19 `StringBuilder city = new StringBuilder("Asheville");`

9.20 The `append` method.

9.21 The `insert` method.

9.22 The `deleteCharAt` method.

9.23 The `setCharAt` method.

9.24 While the `String` class's `replace` method replaces the occurrences of one character with another character, the `StringBuilder` class's `replace` method replaces a specified substring with a string.

9.25 The tokens are "apples", "pears", and "bananas". The delimiter is the space character.

9.26 4 will be stored in `x`.

`"one"` will be stored in `stuff`.

9.27
```
String str = "/home/rjones/mydata.txt"
String[] tokens = str.split("/");
```

9.28
```
String str = "dog$cat@bird%squirrel"
String[] tokens = str.split("[$@%]");
```

9.29 `str = Integer.toString(i);`

9.30 The `toBinaryString`, `toHexString`, and `toOctalString` methods are static members of the `Integer` and `Long` wrapper classes.

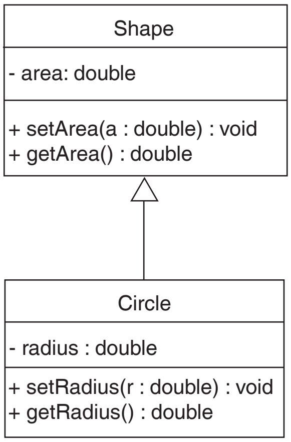
9.31 These variables hold the minimum and maximum values for a particular data type.

## Chapter 10

10.1 `Vehicle` is the superclass and `Truck` is the subclass.

10.2 a. `Shape` is the superclass, `Circle` is the subclass.

b.


Shape
- area: double
+ setArea(a : double) : void
+ getArea() : double
Circle
- radius : double
+ setRadius(r : double) : void
+ getRadius() : double

c. `setArea`, `getArea`, `setRadius`, `getradius`
d. `area`
e. `c.setRadius(10.0);` legal
   `s.setRadius(10.0);` illegal
   `System.out.println(c.getArea());` legal
   `System.out.println(s.getArea());` legal

10.3 The superclass constructor (class `A`) will execute first, then the subclass constructor (class `B`) will execute.

10.4
```
You are on the ground.
You are in the sky.
```

10.5
```
The ground is green
The sky is blue
```

10.6 When the superclass method is inadequate for the subclass's purpose.

10.7 A subclass may call an overridden superclass method by prefixing its name with the `super` key word and a dot (`.`).

10.8 It overrides the superclass method.

10.9 It overloads the superclass method.

10.10 You declare the method with the `final` modifier.

10.11 Protected members of class may be accessed by methods in a subclass, and by methods in the same package as the class.

10.12 Private members may be accessed only by methods in the same class. A protected member of a class may be directly accessed by methods of the same class or methods of a subclass. In addition, protected members may be accessed by methods of any class that are in the same package as the protected member's class.

10.13 Because any other class that inherits from the class, or is in the same package, has unrestricted access to the protected member.

10.14 Private members may be accessed only by methods in the same class. Any method in the same package as the class may directly access the class's members that have package access.

10.15 When you accidentally leave out the access specifier, the member will have package access.

10.16 `ClassD` still inherits from `Object`, because all classes ultimately inherit from `Object`.

10.17 Because those methods are members of the `Object` class.

10.18 a. Legal, because a `Cube` is a `Rectangle`.
b. `System.out.println(r.getLength());` Legal
   `System.out.println(r.getWidth());` Legal
   `System.out.println(r.getHeight());` Illegal
   `System.out.println(r.getSurfaceArea());` Illegal
e. Illegal, because a `Rectangle` is not a `Cube`. The `Cube` class has capabilities beyond those of the `Rectangle` class, so a `Cube` variable cannot reference a `Rectangle` object.

10.19 Abstract methods are used to ensure that a subclass implements the method.

10.20 Override the abstract method.

10.21 An abstract class serves as a superclass for other classes. It represents the generic or abstract form of all the classes that inherit from it.

10.22 An abstract class cannot be instantiated. It must serve as a superclass.

10.23 To specify behavior for other classes.

10.24 It cannot be instantiated.

10.25 An interface only specifies methods; it does not define them. In addition, all members of an interface are public.

10.26 As `final` and `static`.

10.27 `public class Customer implements Relatable`

10.28 `public class Employee implements Payable, Listable`

## Chapter 11

11.1 An exception is an object that is generated as the result of an error or an unexpected event.

11.2 To throw an exception means to generate an exception object.

11.3 Unless an exception is detected by the application and dealt with, it causes the application to halt.

11.4 The `Throwable` class.

11.5 Classes that extend the `Error` class are for exceptions that are thrown when a critical error occurs, such as an internal error in the Java Virtual Machine or running out of memory. An application should not try to handle these exceptions. Exception classes that extend the `Exception` class are general exceptions that an application can handle.

11.6 A try block is one or more statements that are executed and can potentially throw an exception. A catch block is code that appears immediately after a `catch` clause, and is executed in response to a particular exception.

11.7 The program will resume with the code that appears after the entire `try/catch` construct.

11.8 Each exception object has a method named `getMessage` that can be used to retrieve the error message for the exception.

11.9 When an exception is thrown by code in the try block, the JVM begins searching the `try` statement for a `catch` clause that can handle it. It searches the `catch` clauses from top to bottom and passes control of the program to the first `catch` clause with a parameter that is compatible with the exception.

11.10 Statements in the finally block are always executed after the try block has executed and after any catch blocks have executed if an exception was thrown. The finally block executes whether an exception is thrown or not.

11.11 A call stack is an internal list of all the methods that are currently executing. A stack trace is a list of all the methods in the call stack.

11.12 Because method B does not handle the exception, control is passed back to method A. Method A doesn't handle the exception either, so control is passed back to method `main`. Because `main` doesn't handle the exception, the program is halted and the default exception handler handles the exception.

11.13 Unchecked exceptions are those that are extended from the `Error` class or the `RuntimeException` class. A program should not attempt to handle unchecked exceptions. All the remaining exceptions are checked exceptions. Either a checked exception must be handled by the program, or applicable `throws` clauses must be used with all of the methods that can potentially throw the checked exception.

11.14 When code in the method can potentially throw a checked exception, but does not handle the exception.

11.15 The `throw` statement causes an exception object to be created and thrown.

11.16 The argument contains a custom error message that can be retrieved from the exception object's `getMessage` method. If you do not pass a message to the constructor, the exception will have a null message.

11.17 The `throw` statement causes an exception to be thrown. The `throws` clause informs the compiler that a method throws one or more exceptions.

11.18 No. If the method throws an unchecked exception, it does not have to have a `throws` clause.

11.19 You can create a checked exception by extending its class from `Exception`. You can create an unchecked exception by extending its class from `Error`.

11.20 All of the data stored in a text file is formatted as text. Numeric data is converted to text when it is stored in a text file. You can view the data stored in a text file by opening it with a text editor, such as Notepad. In a binary file, data is not formatted as text. Subsequently, you cannot view a binary file's contents with a text editor.

11.21 To write data to a binary file you use the `FileOutputStream` and `DataOutputStream` classes. To read data from a file you use the `FileInputStream` and `DataInputStream` classes.

11.22 With sequential access, when a file is opened for input, its read position is at the very beginning of the file. This means that the first time data is read from the file, the data will be read from its beginning. As the reading continues, the file's read position advances sequentially through the file's contents. In random file access, a program may immediately jump to any location in the file without first reading the preceding bytes.

11.23 `RandomAccessFile`

11.24 The two modes are "r" for reading, and "rw" for reading and writing. When a file is opened with "r" as the mode, the program can only read from the file. When a file is opened with "rw" as the mode, the program can read from the file and write to it.

11.25 The class must implement the `Serializable` interface. You can then use the `ObjectOutputStream` class's `writeObject` method to serialize objects of the class.

## Chapter 12

12.1 A computer's user interface is the part of the computer with which the user interacts.

12.2 A command line interface, which is also known as a console interface, requires the user to type commands. If a command is typed correctly, it is executed and the results are displayed. If a command is not typed correctly, an error message is displayed.

12.3 In a text-based environment, such as a command line interface, programs determine the order in which things happen.

12.4 A program that responds to the user's actions, such as the clicking of a button.

12.5 A standard Java library for developing rich applications that employ graphics.

12.6 A scene

12.7 A tree-like hierarchical data structure that holds the objects in a GUI.

12.8 Root, branch, and leaf.

12.9 Branch nodes can have children. Leaf nodes cannot have children.

12.10 `Application`

12.11 The `launch` method creates a `Stage` object that will be the application's window. It then calls the `start` method, passing a reference to the `Stage` object as an argument.

12.12 `start` is the entry point for the application.

12.13 Set the text that is to be displayed in the window's title var.

12.14 It displays the window.

12.15 An HBox arranges controls in a single horizontal row. A VBox arranges controls in a single vertical row.

12.16 `GridPane`

12.17 `javafx.scene`

12.18 With the `setAlignment` method

12.19 `javafx.geometry`

12.20 `javafx.scene.image`

12.21 An `ImageView` object

12.22 You call the `ImageView` class's `setFitWidth` and `setFitHeight` methods.

12.23 The `ImageView` class's `setPreserveRatio` method.

12.24 You call the `ImageView` class's `setImage` method.

12.25 Spacing is space that appears between controls that are inside the container, and padding is space that appears around the inside edge of the container.

12.26 `javafx.geometry`

12.27 With the container's `setHgap` and `setVgap` methods

12.28 You call the container's `add` method. The arguments are: the control being added, the column, and the row.

12.29 An event is an action that takes place while a program is running. For example, each time the user clicks a Button control, an event takes place.

12.30 An object that responds to events. If an event source is connected to an event handler, a specific method in the event handler is called and the event object is passed as an argument to the method.

12.31 `javafx.event`

12.32 You call the `Button` class's `setOnAction` method.

12.33 `javafx.scene.control`

12.34 You call the control's `getText` method. The method returns the value that has been entered into the TextField as a `String`.

12.35
- Writing the definition of an inner class that implements the `EventHandler` interface. Then, instantiating that class, and registering it with a control.
- Instantiating an anonymous inner class that implements the `EventHandler` interface, and registering the object with a control.
- Using a lambda expression to instantiate an anonymous inner class that implements the `EventHandler` interface, and registering the object with a control.

12.36 Top, bottom, left, center, and right

12.37 Constructor #1: The no-arg constructor creates an empty BorderPane container.

Constructor #2: This constructor accepts one argument. The node that passed as the argument is placed in the BorderPane's center region.

Constructor #3: This constructor accepts five nodes as arguments: one to place in each region.

## Chapter 13

13.1
```
.button
.text-field
.abel
.image-view
```

13.2 `.menu-button`

13.3 `-fx-font-family`

13.4 `-fx-font-weight`

13.5 `-fx-border-style`

13.6 `scene.getStylesheets().add("stylesheet.css");`

13.7 `.root`

13.8 The `.root` definition

13.9 red, green, blue

13.10 blue

13.11 An ID selector

13.12 You add the `RadioButtons` to a `ToggleGroup`. Only one of the `RadioButton` controls in a `ToggleGroup` may be selected at any time.

13.13 The `RadioButton` has a method, `isSelected`, that returns `true` if it is selected, or `false` otherwise.

13.14 Call the `RadioButton`'s `setSelected` method, passing `true` as the argument.

13.15 `ActionEvent`

13.16 The `CheckBox` control has a method, `isSelected`, that returns `true` if it is selected, or `false` otherwise.

13.17 Call the `CheckBox`'s `setSelected` method, passing `true` as the argument.

13.18 `ActionEvent`

13.19 With the control's `setPrefSize` method.

13.20 There are two ways: (1) You can use the `ListView` class's `getSelectionModel().getSelectedItem()` method get the item that is currently selected, or (2) You can use the `ListView` class's `getSelectionModel().getSelectedIndex()` method get the index of the item that is currently selected.

13.21 In single selection mode, only one item can be selected at a time in a ListView control.

13.22 In multiple interval selection mode, multiple items may be selected in a ListView control.

13.23 You can set the orientation to either vertical or horizontal with the `ListView` class's `setOrientation` method. When you call the method, you pass it one of the following `enum` constants: `Orientation.VERTICAL` or `Orientation.HORIZONTAL`.

13.24 `ObservableList`

13.25 You can use the `ComboBox` class's `getValue()` method get the item that is currently selected.

13.26 `ActionEvent`

13.27 An editable `ComboBox` allows the user to select an item from a drop-down list, or type input into a field that is similar to a `TextField`. By default, `ComboBox` controls are uneditable, which means that the user cannot type input into the control; he or she can only select items from a drop-down list.

13.28 a. `setMajorTickUnit(value)`
b. `setMinorTickCount(value)`
c. `setShowTickMarks(value)`
d. `setShowTickLabels(value)`

13.29 Change event

13.30 You call the Slider's `getValue()` method.

13.31 A `TextField` can accept multiple lines of input.

13.32 40 columns. You can change the number of columns with the `setPrefColumnCount()` method.

13.33 10 rows. You can change the number of rows with the `setPrefRowCount()` method.

13.34 When text is entered into the control and the end of a line is reached, the text wraps around to the next line. You can enable text wrapping with the `TextArea` class's `setWrapText` method.

13.35 You call the control's `getText()` method.

13.36

- **Menu Bar.** At the top of the window, is a menu bar. The *menu bar* lists the names of one or more menus.
- **Menu.** A *menu* is a drop-down list of menu items. The user may activate a menu by clicking on its name on the menu bar. In the figure, the *Edit* menu has been activated.
- **Menu Item.** A *menu item* can be selected by the user. When a menu item is selected, some type of action is usually performed.
- **Check menu item.** A *check menu item* appears may be selected or deselected. When it is selected, a check mark appears next to the item. When it is deselected, the check does not appear. Check menu items are normally used to turn an option on or off. The user toggles the state of a check menu item each time he or she selects it.
- **Radio menu item.** A *radio menu item* may be selected or deselected. A radio menu item looks like a check menu item. When it is selected, a check mark appears next to the item. When it is deselected, the check does not appear. When a set of radio menu items are grouped with a `ToggleGroup` object, only one of them can be selected at a time. When the user selects a radio menu item, the one that was previously selected is deselected.
- **Submenu.** A menu within a menu is called a *submenu*. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear.
- **Separator bar.** A separator bar is a horizontal bar used to separate groups of items on a menu. Separator bars are only used as a visual aid and cannot be selected by the user.

13.37 `MenuBar`

13.38 `Menu`

13.39 `MenuItem`

13.40 `RadioMenuItem`. Call the object's `setSelected` method to select it.

13.41 Add them to a `ToggleGroup`.

13.42 `CheckMenuItem`. Call the object's `setSelected` method to select it.

13.43 `ActionEvent`

13.44 `javafx.stage`

13.45 Call the `FileChooser`'s `showOpenDialog` method.

13.46 Call the `FileChooser`'s `showSaveDialog` method.

13.47 The `showOpenDialog` and `showSaveDialog` methods return a `File` object containing information about the selected file.

## Chapter 14

14.1 (0,0)

14.2 (639, 479)

14.3 In the Cartesian coordinate system, the *Y* coordinates decrease as you move downward. In the screen coordinate system, the *Y* coordinates increase as you move downward, toward the bottom of the screen.

14.4 `javafx.scene.shape`

14.5 `javafx.scene.paint`

14.6 `javafx.scene.layout`

14.7 `setStroke()`

14.8 `setFill()`

14.9 `setRotate()`

14.10 `setScaleX()`

14.11 `setScaleY()`

14.12 `Triangle`

14.13 `TranslateTransition`

14.14 `RotateTransition`

14.15 `ScaleTransition`

14.16 `StrokeTransition`

14.17 `FillTransition`

14.18 `FadeTransition`

14.19 With the `Duration` class

14.20 `javafx.util`

14.21 You call the transition class's `setInterpolator` method.

14.22 `javafx.scene.effect`

14.23 `DropShadow`

14.24 `InnerShadow`

14.25 `ColorAdjust`

14.26 `BoxBlur`, `GuassianBlur`, and `MotionBlur`

14.27 `SepiaTone`

14.28 `Glow`

14.29 `Reflection`

14.30 `setInput`

14.31 `Media` and `MediaPlayer`

14.32 `javafx.scene.media`

14.33 If you pass `true` to the method, the sound file will be immediately played once the `MediaPlayer` object is created and ready.

14.34 It specifies the number of times that the sound file should be played.

14.35 `Media`, `MediaPlayer`, and `MediaPlayer`

14.36 `javafx.scene.media`

14.37 With the `MediaView` class's `setFitWidth` and `setFitHeight` methods.

14.38 `KEY_PRESSED`

14.39 `KEY_RELEASED`

14.40 `KEY_TYPED`

14.41 `getCode()`

14.42 The `getCode()` method returns a `KeyCode` value, which is an `enum` in the `javafx.scene.input` package. The `KeyCode` enum contains an extensive set of constants representing the keys on a computer keyboard.

## Chapter 15

15.1 A recursive algorithm requires multiple method calls. Each method call requires several actions to be performed by the JVM. These actions include allocating memory for parameters and local variables, and storing the address of the program location where control returns after the method terminates. All of these actions are known as overhead. In an iterative algorithm, which uses a loop, such overhead is unnecessary.

15.2 A case in which the problem can be solved without recursion.

15.3 Cases in which the problem is solved using recursion.

15.4 When it reaches the base case.

15.5 In direct recursion, a recursive method calls itself. In indirect recursion, method A calls method B, which in turn calls method A.

## Chapter 16

16.1 Bubble sort

16.2 Insertion sort

16.3 Selection sort

16.4 Quicksort

16.5 The sequential search algorithm simply uses a loop to step through each element of an array, comparing each element's value with the value being searched for. The binary search algorithm, which requires the values in the array to be sorted in order, starts searching at the element in the middle of the array. If the middle element's value is greater than the value being searched for, the algorithm next tests the element in the middle of the first half of the array. If the middle element's value is less than the value being searched for, the algorithm next tests the element in the middle of the last half of the array. Each time the array tests an array element and does not find the value being searched for, it eliminates half of the remaining portion of the array. This method continues until the value is found, or there are no more elements to test. The binary search is more efficient than the sequential search.

16.6 10,000

16.7 15

16.8 The items frequently searched for can be stored near the beginning of the array.

16.9 A basic operation is one that requires constant time, regardless of the size of the problem that is being solved.

16.10 The worst case complexity function $f(n)$ of an algorithm is a measure of the time required by the algorithm to solve a problem instance of size $n$ that requires the most time.

16.11 Because $100n$ and $25n$ differ by a constant factor and constant factors are not significant, the two algorithms are equivalent in efficiency.

16.12 To say that $f(n)$ is not in $O(g(n))$ means that $f(n)$ cannot be bounded by any constant multiple of $g(n)$.

16.13 To say that $f(n)$ is in $O(g(n))$ means that there is a positive constant $K$ such that $f(n) \leq Kg(n)$ for all $n \geq 1$. This means that for large problem sizes, an algorithm with complexity function $f(n)$ is no worse than one with complexity function $g(n)$.

16.14 To show that $100n^3 + 50n^2 + 75$ is in $O(20n^3)$, observe that

$$\frac{100n^3 + 50n^2 + 75}{20n^3} = 5 + \frac{5}{2n} + \frac{75}{20n^3} \leq 5 + 5 + 75 \leq 85$$

for all $n \geq 1$. Take $K = 85$ in Equation 17.1.

16.15 The problem definition and algorithm are as follows:

MAIN ALGORITHM
INPUT: two integer arrays *a*[ ] and *b*[ ] of size *n*
SIZE OF INPUT: The number *n* of array entries
OUTPUT: true if each element of *a*[ ] is also an element of *b*[ ], false otherwise

```
boolean contained = true
int i = 0
While contained && i < n do
   contained = LINSEARCH(a[i], b)
   i++;
End While
print contained
```

SEARCH ALGORITHM
INPUT: an integer *X* and an array *b*[ ] of size *n*
SIZE OF INPUT: The number *n* of array entries
RETURN: true if *X* is contained in *b*[ ], false otherwise

```
boolean LINSEARCH(int X, int [ ] b)
Begin
   int k = 0;
   boolean found = false;
   While !found && k < n do
      If (X == b[k])
         found = true;
      else
         k++
      END If
   End While
   return found
End LINSEARCH
```

16.16 See solution of 16.15 for the algorithm. In the worst case, the main algorithm makes $n$ calls to the `LINSEARCH` procedure. Each call to `LINSEARCH` makes at most $n^2$ comparisons. Therefore, the algorithm makes a total of comparisons.

16.17 Let $f(n)$ be in O(20). Then there is a positive $K$ such that $f(n) \leq 20K$ for all $n \geq 1$. Let $K_1 = 20K$. Then $f(n) \leq 1K_1$ for all $n \geq 1$, so $f(n)$ is in O(1).

16.18 Assuming that $g(n) \geq 1$ for all $n \geq 1$, we have $100 \leq 100g(n)$ for all $n \geq 1$. This implies that $g(n) + 100 \leq g(n) + 100g(n) = 101g(n)$ for all $n \geq 1$. Now if $f(n)$ is in $O(g(n) + 100)$, there exists a positive $K$ such that $f(n) \leq K(g(n) + 100) \leq 101Kg(n)$ for all $n \geq 1$. Taking $K_1 = 101K$, we see that $f(n) \leq K_1 g(n)$ for all $n \geq 1$.

# Chapter 17

17.1 Because it opens an opportunity for the application to throw an exception at run time. If we retrieve an object from the `ArrayList`, but cast it to the wrong type, an exception occurs.

17.2 In non-generic mode, the `get` method returns an `Object` reference. When the `get` method is called, the compiler does not know the exact type of the object being returned, only that it is an `Object`. To assign the reference to any other type of variable we have to cast it to the appropriate type.

17.3 `Rectangle` objects.

17.4 Compile time.

17.5 A type parameter is an identifier in a generic class declaration. It represents an actual type that will be used when a generic class is instantiated.

17.6 Only reference types may be passed to a type parameter.

17.7 Yes, it is possible. Doing so instantiates the class as a raw type. This is not recommended, however, because you give up the benefits of type-safety that you would have otherwise.

17.8 No, it is not required, but it is the standard convention.

17.9 That any type argument can be used in its place.

17.10 It means that any type that extends `Number`, or is `Number`, can be passed as a type argument.

17.11 It means that any type that is `Integer` or a superclass of `Integer` can be passed as a type argument.

17.12 An upper boundary

17.13 A lower boundary

17.14 No, you cannot use both together.

17.15 When you call a generic method, the compiler determines which type to use from the context in which you are using the method.

17.16 To prevent the class from being instantiated with a type argument that does not support some operation that the class is performing.

17.17 A generic class's type can be constrained with either an upper bound or a lower bound.

17.18 A generic class can be a superclass and/or a subclass.

17.19 Yes. The "is-a" relationship is in effect between the classes.

17.20 Yes.

17.21 Yes.

17.22 You would write the type parameter definition as:

```
<T extends Comparable>
```

17.23 When the compiler encounters a generic class, interface, or method with an unbound type parameter, such as `<T>` or `<E>`, it replaces all occurrences of the type parameter with `Object`.

17.24 When the compiler encounters a class, interface, or method with a bound type parameter, such as `<T extends Number>` or `<E extends Comparable>`, it replaces all occurrences of the type parameter with the bound that is applied to the parameter.

17.25 No.

## Chapter 18

18.1 The `java.util` package.

18.2 Lists, sets, and maps.

18.3 Elements are stored in a list according to position, and a list can have duplicate values. A set has no concept of position and does not allow duplicates.

18.4 Each element of a map consists of a key associated with a value.

18.5 `List` and `Set` extend the `Collection` interface.

18.6 `forEach()` and `iterator()`.

18.7 An iterator is an object that can move through a collection and provide access to each element of that collection.

18.8 Contiguous allocation.

18.9 Linked allocation.

18.10 So you can change the class that implements the interface without having to change the rest of the code.

18.11 The `Iterator` interface.

18.12 A `ListIterator` object can move forward and backward through a collection, while an `Iterator` object can only move forward.

18.13 Call the `listIterator()` method.

18.14 By calling the `hasNext()` method.

18.15 The `hasPrevious()` method.

18.16 The element returned by the last call to the `next()` method.

18.17 The element returned by the last call to `next()` or `previous()`.

18.18 The capacity is the length of the array that underlies the `ArrayList`, while the size is the number of elements currently stored.

18.19 The `LinkedList` class.

18.20 The `LinkedList` class.

18.21 A hash code of an object is an integer value that is characteristic of that object.

18.22 The `hashCode()` method.

18.23 A collision occurs when two different objects hash to the same value.

18.24 The duplicate item is not added to the set.

18.25 The `hashCode()` method.

18.26 The `LinkedHashSet` class.

18.27 The `TreeSet` class.

18.28 A comparator for a class is an object that can compare two objects of that class and determine whether they are equal, and if not, which comes before the other. A comparator implements the `Comparator` interface.

18.29 Each element of map consists of a key and a value.

18.30 The key.

18.31 Insertion order and access order.

18.32 By key order.

18.33 A stream is an object that permits a pipeline of operations to be performed on a sequence of elements drawn from a source.

18.34 An intermediate operation transforms a stream into another stream, while a terminal operation transforms a stream into a non-stream object or result.

18.35 The `Stream` interface, the Collection interface, and the `Arrays` class all have methods for creating streams.

18.36 A pipeline is a sequence of operations where the output of one operation becomes the input to the next operation in the sequence.

18.37 A terminal operation and a reduction are the same thing.

18.38 A collector is an object that performs a reduction on a stream.

18.39 A stream pipeline can have at most one reduction, which must be the last operation in the pipeline.

## Chapter 19

19.1

```
Node greenNode  = new Node("green");
greenNode.next = new Node("blue");
Node redNode = new Node("red", greenNode);
```

19.2 `BrownYellowPurple`

19.3

```
// Let p point to the new Node
Node p = new Node(e, succ, succ.prev);
succ.prev = p;
if (p.prev == null)
   first = p;
else
   p.prev.succ = p;
```

19.4

```
last.next  = new Node(e, null, last);
last = last.next;
```

## Chapter 20

20.1 push

20.2 pop

20.3 Throw an exception

20.4 Last in first out

20.5 Some examples are a stack of plates or trays in a cafeteria, a line of cars parked single file in a narrow driveway, and a person wearing an undershirt, a shirt, a jacket, and an overcoat.

20.6 enqueue

20.7 First in first out

20.8 Some examples are a grocery store checkout line, a line of cars going through a toll booth.

20.9 By the time the add operation reaches the end of the array, the remove operation may have emptied the spaces of the beginning of the array, allowing the add operation to wrap around and start storing elements at the beginning. Similarly, when the remove empties spaces at the end of the array, it should wrap around to the beginning.

## Chapter 21

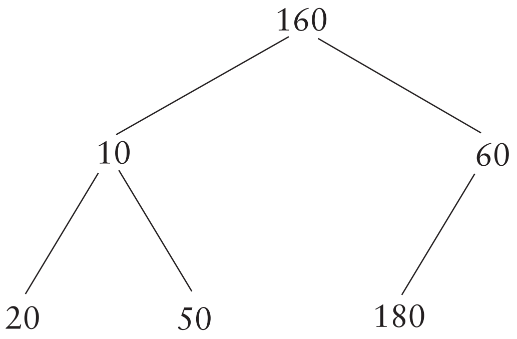
21.1
- a. Ancestors of $A$ : $\{A\}$; ancestors of $B$: $\{B, A\}$; ancestors of $C$: $\{C, A\}$; ancestors of $D$: $\{D, B, A\}$ : ancestors of $E$: $\{E, C, A\}$; ancestors of $F$: $\{F, C, A\}$; ancestors of $G$: $\{G, F, C, A\}$.
- b. Descendants of $A$: $\{A, B, D\}$; descendants of $B$: $\{B, D\}$; descendants of $D$: $\{D\}$; descendants of $C$: $\{C, E, F, G\}$; descendants of $E$: $\{E\}$; descendants of $F$: $\{F, G\}$; descendants of $G$: $\{G\}$.
- c. Leaves of T: $\{D, E, G\}$.
- d. Non-leaf nodes of $T$: $\{A, B, C, F\}$.
- e. Preorder: $\{A, B, D, C, E, F, G\}$.
- f. Postorder: $\{D, B, E, G, F, C, A\}$.
- g. Inorder: $\{B, D, A, E, C, G, F\}$.

21.2 The tree on the left : $x\ 12\ +$
`The tree on the right:` $x\ 12 + y * 2\ +$

21.3 The tree on the left: $+\ x\ 12$
`The tree on the right:` $+ * + x\ 12\ y\ 2$

21.4 Nodes in preorder: 160 10 20 50 60 180

Drawing of the tree:



## Chapter 22 (Available on the Book's Companion Website)

22.1 Traditional text and binary files are not practical when a large amount of data must be stored and manipulated. Many businesses keep hundreds of thousands, or even millions, of data items in files. When a text or binary file contains this much data, simple operations such as searching, inserting, and deleting become cumbersome and inefficient.

22.2 The DBMS handles the actual reading, writing, and searching of data. The Java application programmer needs only to know how to communicate with the DBMS.

22.3 JDBC, or Java Database Connectivity

22.4 SQL, or Structured Query Language

22.5 A string listing the protocol that should be used to access a database, the name of the database, and potentially other items

22.6 `jdbc:derby:InventoryDB`

22.7 `DriverManager.getConnection`

22.8 The data that is stored in a table is organized into rows and columns. A row is a complete set of information about a single item. The data that is stored in a row is divided into columns. Each column is an individual piece of information about the item.

22.9 A primary key is a column that holds a unique value for each row, and can be used to identify specific rows.

22.10
- `int`
- `int`
- `float`
- `String`
- `String`
- `String`
- `double`

22.11 A `ResultSet` is an object that contains the results of an SQL statement.

22.12
a. The table is `Account`.
b. The column is `Id`.

22.13
a. `SELECT * FROM Inventory`
b. `SELECT ProductID FROM Inventory`
c. `SELECT ProductID, QtyOnHand FROM Inventory`
d. `SELECT ProductID FROM Inventory`
   `WHERE Cost < 17.00`
e. `SELECT * FROM Inventory`
   `WHERE ProductID LIKE '%ZZ'`

22.14 The `LIKE` operator can be used in a search criterion to search for a string that contains a pattern of characters.

22.15 The `%` symbol represents any sequence of 0 or more characters. The underscore character ( _ ) represents only a single character.

22.16 You use the `ORDER BY` clause.

23.17

```
Connection conn = DriverManager.getConnection(DB_URL);
Statement stmt = conn.createStatement();
ResultSet result = stmt.executeQuery(sql);
stmt.close();
conn.close();
```

22.18 You create a `Statement` object (`Statement` in an interface in `java.sql`), and you pass a string containing the `SELECT` statement to the `Statement` object's `executeQuery` method.

22.19 The cursor initially points before the first row. To move it forward you call the `ResultSet` object's `next` method.

22.20 You call the `ResultSet` object's `getString` method, passing 3 as an argument.

22.21

```
INSERT INTO Coffee
   (Description, ProdNum, Price)
VALUES
   ('Eastern Blend', '30-001', 18.95)
```

22.22

```
INSERT INTO Coffee
   (Description, ProdNum, Price)
VALUES
   ('Honduran Dark', '22-001', 8.65)
```

22.23

```
UPDATE Coffee
SET Price = 4.95
WHERE Description Like '%Decaf%'
```

22.24

```
DELETE FROM Coffee WHERE Description LIKE '%Decaf%'
```

22.25

```
CREATE TABLE Book
( Publisher CHAR(25),
Author, CHAR(25),
Pages INTEGER,
Isbn CHAR(10) )
```

22.26

```
DROP TABLE Book
```

## Chapter 23 (Available on the Book's Companion Website)

23.1 A frame is a basic window that has a border around it, a title bar, and a set of buttons for minimizing, maximizing, and closing the window. In a Swing application, you create a frame from the `JFrame` class.

23.2 With the `setSize` method.

23.3 With the `setVisible` method.

23.4 A content pane is a container that is part of every `JFrame` object. You cannot see the content pane and it does not have a border, but any component that is added to a `JFrame` must be added to its content pane.

23.5 Panels cannot be displayed by themselves.

23.6 It is an object that responds to events.

23.7 The class must implement the `ActionListener` interface, and it must have a method named `actionPerformed`. The method is executed when the user clicks the button.

23.8 With the `addActionListener` method.

23.9 You use the `setBackground` method to set the color of a component, and the `setForeground` method to set the color of text displayed on a component.

23.10 By calling the container's `setLayout` method and passing a reference to a layout manager object as the argument.

23.11 `BorderLayout`

23.12 `FlowLayout`

23.13 `GridLayout`

23.14 Only one, in both cases.

23.15 By placing the component in a panel, then adding the panel to the region. The layout manager resizes the panel, not the component inside the panel.

23.16 By calling the `pack` method.

23.17 `BorderLayout` is the default layout manager for a `JFrame` object's content pane.

`FlowLayout` is the default layout manager for a `JPanel` object.

23.18 Radio buttons

23.19 Check boxes

23.20 A `ButtonGroup` object can contain radio buttons. It creates a mutually exclusive relationship between the radio buttons that it contains.

23.21 Radio buttons.

23.22 An action event.

23.23 An item event.

23.24 You use the `isSelected` method to determine whether a `JRadioButton` component is selected. The method returns a `boolean` value. If the check box is selected, the method returns `true`. Otherwise, it returns `false`.

23.25 You use the `isSelected` method to determine whether a `JCheckBox` component is selected. The method returns a `boolean` value. If the check box is selected, the method returns `true`. Otherwise, it returns `false`.

23.26 `setBorder`

23.27 You should use the `BorderFactory` class to create them for you. The `BorderFactory` class has static methods that return various types of borders.

## Chapter 24 (Available on the Book's Companion Website)

24.1 You call its `setEditable` method and pass `false` as the parameter. To store text in the text field, use the `setText` method.

24.2 The index of the first item is 0. The index of the twelfth item would be 11.

24.3 You use the `getSelectedValue` method to retrieve the selected item, and the `getSelectedIndex` method to get the index of the selected item.

24.4 First, set the number of visible rows for the list component. Next, create a scroll pane object and add the list component to it.

24.5 You use the `getSelectedItem` method to retrieve the selected item, and the `getSelectedIndex` method to get the index of the selected item.

24.6 An uneditable combo box combines a button with a list and allows the user to select only items from its list. An editable combo box combines a text field and a list. In addition to selecting items from the list, the user may also type input into the text field. The default type of combo box is uneditable.

24.7 To display an image, you first create an instance of the `ImageIcon` class, which can read the contents of an image file. Next, you display the image in a label by passing the `ImageIcon` object as an argument to the `JLabel` constructor. You can create a `JLabel` with both an image and text by creating the `JLabel` in the usual way, with the text passed to the constructor. Then, you can use the `setIcon` method to display an image.

24.8 To display an image, you first create an instance of the `ImageIcon` class, which can read the contents of an image file. Next, you display the image in a button by passing the `ImageIcon` object as an argument to the `JButton` constructor. To create a button with an image and text, pass a `String` and an `ImageIcon` object to the constructor.

24.9 `setIcon`

24.10 A mnemonic is a key on the keyboard that you press in combination with the Alt key to quickly access a component such as a button. These are sometimes referred to as short-cut keys, or hot keys. You assign an access key to a component through the component's `setMnemonic` method.

24.11 A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component. The box usually gives a short description of what the component does. You assign a tool tip to a component with the `setToolTipText` method.

24.12
- a. **Menu bar.** At the top of the window, just below the title bar, is a menu bar. The menu bar lists the names of one or more menus.
- b. **Menu item.** A menu item can be selected by the user. When a menu item is selected, some type of action is usually performed.
- c. **Check box menu item.** A check box menu item appears with a small box beside it. The item may be selected or deselected. When it is selected, a check mark appears in the box. When it is deselected, the box appears empty. Check box menu items are normally used to turn an option on or off. The user toggles the state of a check box menu item each time he or she selects it.
- d. **Radio button menu item.** A radio button menu item may be selected or deselected. A small circle appears beside it that is filled in when the item is selected and appears empty when the item is deselected. Like a check box menu item, a radio button menu item can be used to turn an option on or off. When a set of radio button menu items are grouped with a `ButtonGroup` object, only one of them can be selected at a time. When the user selects a radio button menu item, the one that was previously selected is deselected.
- e. **Submenu.** A menu within a menu is called a submenu. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear.
- f. **Separator bar.** A separator bar is a horizontal bar used to separate groups of items on a menu. Separator bars are used only as a visual aid and cannot be selected by the user.

24.13 The `JMenuItem` class. You pass a string, which is displayed on the menu item.

24.14 The `JRadioButtonMenuItem` class. You pass a string to the constructor, which is displayed on the menu item. To cause it to be initially selected, you pass `true` as an optional second argument to the constructor.

24.15 You add them to a `ButtonGroup`.

24.16 The `JCheckBoxMenuItem` class. You pass a string to the constructor, which is displayed on the menu item. To cause it to be initially selected, you pass `true` as an optional second argument to the constructor.

24.17 The `JMenu` class. You pass a string to the constructor, which is displayed on the menu.

24.18 The `JMenuBar` class.

24.19 With the `JFrame` object's `setJMenuBar` method.

24.20 Action events.

24.21 With the `setPreferredSize` method.

24.22 The first argument to the `Dimension` class constructor is the component's width, and the second argument is the component's height.

24.23 One constructor accepts the size of the text area, in rows and columns. Another constructor accepts the string that is to be displayed initially in the text area, as well as the size of the text area in rows and columns.

24.24 With the `getText` method.

24.25 `JTextArea` components do not automatically display scroll bars. To display scroll bars on a `JTextArea` component, you must add it to the scroll pane.

24.26 Line wrapping causes text to automatically wrap to the next line when a line is filled. There are two different styles of line wrapping: word wrapping and character wrapping. When *word wrapping* is performed, the line breaks always occur between words, never in the middle of a word. When character wrapping is performed, lines are broken between characters. You use the `JTextArea` class's `setLineWrap` method to turn line wrapping on. The method accepts a `boolean` argument. If you pass `true`, then line wrapping is turned on. If you pass `false`, line wrapping is turned off. You specify the style of line wrapping that you prefer with the `JTextArea` class's `setWrapStyleWord` method. This method accepts a `boolean` argument. If you pass `true`, then the text area will perform word wrapping. If you pass `false`, the text area will perform character wrapping.

24.27 A `Font` object.

24.28 The first argument is the name of a font. The second argument is an `int` that represents the style of the text. The third argument is the size of the text in points.

24.29 It generates a change event.

24.30 a. `setMajorTickSpacing`
b. `setMinorTickSpacing`
c. `setPaintTicks`
d. `setPaintLabels`

## Chapter 25 (Available on the Book's Companion Website)

25.1 When a user accesses a Web page with his or her browser, any applet associated with the Web page is transmitted to the user's system. The applet is then executed on the user's system.

25.2 To prevent malicious code from attacking or spying on unsuspecting users.

25.3 `<html>` marks the beginning and `</html>` marks the end.

25.4 `<head></head>`

25.5 You would use: `<title>My Web Page</title>`
This would go in the document head section.

25.6 `<body></body>`

25.7 `<h1>Student Roster</h1>`

25.8 `<center><b>My Resume</b></center>`

25.9 `<b><i>Hello World</i></b>`

25.10 The `<br />` tag causes a line break. The `<p />` tag causes a paragraph break. The `<hr />` tag displays a horizontal rule.

25.11 `<a href=`
`"http://java.sun.com">Click Here</a>`

25.12 `JApplet`

25.13 `init`

25.14 Because the browser creates an instance of the class automatically.

25.15 `<applet code="MyApplet.class" width=400 height=200>`

25.16 `Applet`

25.17 You simply add them to the `Frame` or `Applet` object. These classes have an `add` method.

25.18 You should override the `paint` method.

25.19 You override the paintComponent method.

25.20 These methods are automatically called when the component is first displayed and are called again any time the component needs to be redisplayed.

25.21 The superclass's `paint` or `paintComponent` method should be called.

25.22 Call the `repaint` method.

25.23 A rectangle.

25.24 The first array contains the *X* coordinates for each vertex, and the second array contains the *Y* coordinates for each vertex.

25.25 a. `drawLine`
b. `fillRect`
c. `fillOval`
d. `fillArc`
e. `setColor`
f. `drawRect`
g. `drawOval`
h. `drawArc`
i. `drawString`
j. `setFont`

25.26 A mouse press event indicates that the mouse button was pressed. A mouse click event indicates that the mouse button was pressed, and then released.

25.27 To handle a mouse click event: `MouseListener`
To handle a mouse press event: `MouseListener`
To handle a mouse dragged event: `MouseMotionListener`
To handle a mouse release event: `MouseListener`
To handle a mouse move event: `MouseMotionListener`

25.28 They accept `MouseEvent` objects. The `getX` and `getY` methods return the mouse's *X* and *Y* coordinates.

25.29 No, they cannot be left out. Empty definitions can be written, or an adapter class can be used.

25.30 An adapter class implements an interface and provides empty definitions for all of the required methods. When you extend a class from an adapter class, it inherits the empty methods. In your subclass, you can override the methods you want and forget about the others.

25.31 Action events

25.32 In milliseconds.

25.33 By calling its `start` method.

25.34 By calling its `stop` method.

25.35 The `play` method.

25.36 The `Applet` class's `play` method loads a sound file, plays it one time, and then releases it for garbage collection. If you need to load a sound file to be played multiple times, you should use an `AudioClip` object. An `AudioClip` object loads a sound file and retains it in memory. It provides methods for playing, looping, and stopping the audio play.

25.37 The `play` method plays the sound file. The `loop` method plays the sound file repeatedly. The `stop` method stops the sound file from playing.

25.38 The `getDocumentBase` method returns a `URL` object containing the location of the HTML file that invoked the applet. The `getCodeBase` method returns a URL object containing the location of the applet's `.class` file.

APPENDIX

# M Answers to Odd-Numbered Review Questions

## Chapter 1

### Multiple Choice

1. b
3. a
5. b
7. c
9. a
11. a
13. b

### Find the Error

1. The algorithm performs the math operation at the wrong time. It multiplies width by length before getting values for those variables.

### Algorithm Workbench

1.
```
Display "What is the customer's maximum amount of credit?"
Input maxCredit.
Display "What is the amount of credit used by the customer?"
Input creditUsed.
$availableCredit = maxCredit - creditUsed$.
Display availableCredit.
```

3.
```
Display "What is the account's starting balance?"
Input startingBalance.
Display "What is the total amount of the deposits made?"
Input deposits.
Display "What is the total amount of the withdrawals made?"
Input withdrawals.
```

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Data Structures*, Fourth Edition. Tony Gaddis. Copyright © 2019 by Pearson Education, Inc. All rights reserved.

```
Display "What is the monthly interest rate?"
Input interestRate.
balance = startingBalance + deposits − withdrawals.
interest = balance * interestRate.
balance = balance + interest.
Display balance.
```

### Predict the Result

1. 7

### Short Answer

1. Main memory, or RAM, holds the sequences of instructions in the programs that are running and the data those programs are using. Main memory, or RAM, is usually volatile. Secondary storage is a type of memory that can hold data for long periods of time—even when there is no power to the computer.
3. An operating system is a set of programs that manages the computer's hardware devices and controls their processes. Windows and UNIX are examples of operating systems. Application software refers to programs that make the computer useful to the user. These programs solve specific problems or perform general operations that satisfy the needs of the user. Word processing, spreadsheet, and database packages are all examples of application software.
5. Because machine language programs are streams of binary numbers, and high-level language programs are made up of words.
7. *Syntax errors* are mistakes that the programmer has made that violate the rules of the programming language. *Logical errors* are mistakes that cause the program to produce erroneous results.
9. A program that translates source code into executable code.
11. Machine language code is executed directly by the CPU. Byte code is executed by the JVM.
13. Object-oriented programming
15. The object's methods.
17. No
19. `javac LabAssignment.java`

## Chapter 2

### Multiple Choice and True/False

1. c
3. a
5. a, c, and d

7. c
9. a
11. b
13. a
15. a
17. True
19. True
21. False

## Predict the Output

1.
```
0
100
```

3.
```
I am the incrediblecomputing
machine
and I will
amaze
you.
```

5.
```
23
1
```

## Find the Error

- The comment symbols in the first line are reversed. They should be `/*` and `*/`.
- The word `class` is missing in the second line. It should read `public class MyProgram`.
- The `main` header should not be terminated with a semicolon.
- The fifth line should have a left brace, not a right brace.
- The first four lines inside the `main` method are missing their semicolons.
- The comment in the first line inside the `main` method should begin with forward slashes (`//`), not backward slashes.
- The last line inside the `main` method, a call to `println`, uses a string literal, but the literal is enclosed in single quotes. It should be enclosed in double quotes, like this: `"The value of c is"`.
- The last line inside the `main` method passes `C` to `println`, but it should pass `c` (lowercase).
- The class is missing its closing brace.

## Algorithm Workbench

1. `double temp, weight, age;`

3. a. `b = a + 2;`

b. `a = b * 4;`

c. `b = a / 3.14;`

d. `a = b - 8;`

e. `c = 'K';`

f. `c = 66;`

5. a. `3.287E6`

   b. `-9.7865E12`

   c. `7.65491E-3`

7. `10 20 1`

9. `a`

11.
```
int speed, time, distance;
speed = 20;
time = 10;
distance = speed * time;
System.out.println(distance);
```

13.
```
double income;
// Create a Scanner object for keyboard input.
Scanner keyboard = new Scanner(System.in);
// Ask the user to enter his or her desired income
System.out.print("Enter your desired annual income: ");
income = keyboard.nextDouble();
```

15. `total = (float)number;`

### Short Answer

1. Multi-line style

3. A self-documenting program is written in such a way that you get an understanding of what the program is doing just by reading its code.

5. The `print` and `println` methods are members of the `out` object. The `out` object is a member of the `System` class. The `System` class is part of the Java API.

7. You should always choose names for your variables that give an indication of what they are used for. The rather nondescript name, `x`, gives no clue as to what the variable's purpose is.

9. In both cases you are storing a value in a variable. An assignment statement can appear anywhere in a program. An initialization, however, is part of a variable declaration.

11. Programming style refers to the way a programmer uses spaces, indentations, blank lines, and punctuation characters to visually arrange a program's source code. An inconsistent programming style can create confusion for a person reading the code.

13. `javadoc SalesAverage.java`

# Chapter 3

## Multiple Choice and True/False

1. b
3. a
5. c
7. a
9. a
11. a
13. c
15. c
17. True
19. True
21. True

## Find the Error

1. Each `if` clause is prematurely terminated by a semicolon.
3. The conditionally executed blocks of code should be enclosed in braces.
5. The `!` operator is only applied to the variable `x`, not the expression. The code should read:
   ```
   if (!(x > 20))
   ```
7. The statement should use the `||` operator instead of the `&&` operator.
9. The `equalsIgnoreCase` method should be used instead of the `equals` method.

## Algorithm Workbench

1.
```
if (y == 0)
x = 100;
```

3.
```
if (sales < 10000)
   commission = .10;
else if (sales <= 15000)
   commission = .15;
else
   commission = .20;
```

5.
```
if (amount1 > 10)
{
   if (amount2 < 100)
   {
      if (amount1 > amount2)
```

```
      {
         System.out.println(amount1);
      }
      else
      {
         System.out.println(amount2);
      }
   }
}
```

7.

```
if (temperature >= -50 && temperature <= 150)
   System.out.printn("The number is valid.");
```

9.

```
if (title1.compareTo(title2) < 0)
   System.out.println(title1 + " " + title2);
else
   System.out.println(title2 + " " + title1);
```

11. C, A, B

13.

```
System.out.printf("%,.2f", number);
```

### Short Answer

1. Conditionally executed code is executed only under a condition, such as an expression being true.
3. By indenting the conditionally executed statements, you are causing them to stand out visually. This is so you can tell at a glance what part of the program the `if` statement executes.
5. A flag is a `boolean` variable that signals when some condition exists in the program. When the flag variable is set to false, it indicates the condition does not yet exist. When the flag variable is set to true, it means the condition does exist.
7. It takes two `boolean` expressions as operands and creates a boolean expression that is true only when both subexpressions are true.
9. They determine whether a specific relationship exists between two values. The relationships are greater-than, less-than, equal-to, not equal-to, greater-than or equal-to, and less-than or equal-to.

## Chapter 4

### Multiple Choice and True/False

1. a
3. c
5. a
7. b
9. c

11. a
13. a
15. d
17. d
19. True
21. False
23. False
25. True

## Find the Error

1. The conditionally executed statements should be enclosed in a set of braces. Also, the `again` variable should be initialized with either 'y' or 'Y'.

3. The expression being tested by the `do-while` loop should be `choice == 1`. Also, the `do-while` loop must be terminated by a semicolon.

## Algorithm Workbench

1.

```
Scanner keyboard = new Scanner(System.in);
int product = 0, num;
while (product < 100)
{
   num = keyboard.nextInt();
   product = num * 10;
}
```

3. The following code simply prints the numbers, separated by spaces.

```
for (int x = 0; x <= 1000; x += 10)
   System.out.print(x + " ");
```

The following code prints the numbers separated by commas.

```
for (int x = 0; x <= 1000; x += 10)
{
   if (x < 1000)
      System.out.print(x + ", ");
   else
      System.out.print(x);
}
```

5.

```
double total = 0;
for (int num = 1, denom = 30; num <= 30; num++, denom--)
   total += num / denom;
```

7.

```
Scanner keyboard = new Scanner(System.in);
int x;
do
```

```
{
   System.out.print("Enter a number: ");
   x = keyboard.nextInt();
} while (x > 0);
```

9.
```
for (int count = 0; count > 50; count++)
   System.out.println("count is " + count);
```

11.
```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number in the range " +
                 "of 1 through 4: ");
number = keyboard.nextInt();
while (number < 1 || number > 4)
{
   System.out.print("Invalid number. Enter a " +
                    "number in the range " +
                    "of 1 through 4: ");
   number = keyboard.nextInt();
}
```

13.
```
for (int r = 7; r > 0; r--)
{
   for (int c = 0; c < r; c++)
   {
      System.out.print("*");
   }
   System.out.println();
}
```

15.
```
import java.util.Random;
public class ReviewQuestion15
{
   public static void main(String[] args)
   {
      Random rand = new Random();
      System.out.println(rand.nextInt(10) + 1);
   }
}
```

17.
```
PrintWriter outFile = new PrintWriter("NumberList.txt");
for (int i = 1; i <= 100; i++)
   outFile.println(i);
outFile.close();
```

19.
```
File file = new File("NumberList.txt");
Scanner inFile = new Scanner(file);
String input;
int number, total = 0;
```

```
while (inFile.hasNext())
{
   input = inFile.nextLine();
   number = Integer.parseInt(input);
   total += number;
}
System.out.println("The total is " + total);
inFile.close();
```

### Short Answer

1. In postfix mode the operator is placed after the operand. In prefix mode the operator is placed before the variable operand. Postfix mode causes the increment or decrement operation to happen after the value of the variable is used in the expression. Prefix mode causes the increment or decrement to happen first.
3. A pretest loop tests its test expression before each iteration. A posttest loop tests its test expression after each iteration.
5. The `while` loop is a pretest loop and the `do-while` loop is a posttest loop.
7. The `do-while` loop.
9. An accumulator is used to keep a running total of numbers. In a loop, a value is usually added to the current value of the accumulator. If it is not properly initialized, it will not contain the correct total.
11. There are many possible examples. A program that asks the user to enter a business's daily sales for a number of days, and then displays the total sales is one example.
13. Sometimes the user has a list of input values that is very long, and doesn't know the number of items there are. When the sentinel value is entered, it signals the end of the list, and the user doesn't have to count the number of items in the list.
15. There are many possible examples. One example is a program that asks for the average temperature for each month, for a period of five years. The outer loop would iterate once for each year and the inner loop would iterate once for each month.
17. Closing a file writes any unsaved data remaining in the file buffer.
19. After the `println` method writes its data, it writes a newline character. The `print` method does not write the newline character.
21. The file does not exist.
23. You create an instance of the `FileWriter` class to open the file. You pass the name of the file (a string) as the constructor's first argument, and the `boolean` value `true` as the second argument. Then, when you create an instance of the `PrintWriter` class, you pass a reference to the `FileWriter` object as an argument to the `PrintWriter` constructor. The file will not be erased if it already exists and new data will be written to the end of the file.

## Chapter 5

### Multiple Choice and True/False

1. b
3. a
5. b
7. b
9. c
11. True
13. False
15. True
17. False
19. False

### Find the Error

1. The header should not be terminated with a semicolon.
3. The method should have a `return` statement that returns a `double` value.

### Algorithm Workbench

1. `doSomething(25);`
3. The value 3 will be stored in `a`, 2 will be stored in `b`, and 1 will be stored in `c`.
5. `result = cube(4);`
7. 
```
public static double timesTen(double num)
{
   return num * 10.0;
}
```
9. 
```
// Assume java.util.Scanner has been imported.
public static String getName()
{
   String name;
   Scanner keyboard = new Scanner(System.in);
   System.out.print("Enter your first name: ");
   name = keyboard.nextLine();
   return name;
}
```

### Short Answer

1. A large complex problem is broken down into smaller manageable pieces. Each smaller piece of the problem is then solved.

3. An argument is a value that is passed into a method when the method is called. A parameter variable is a variable that is declared in the method header, and receives the value of an argument when the method is called.

5. When an argument is passed to a method, only a copy of the argument is passed. The method cannot access the actual argument.

## Chapter 6

### Multiple Choice and True/False

1. a
3. d
5. b
7. d
9. b
11. c
13. b
15. True
17. False
19. False

### Find the Error

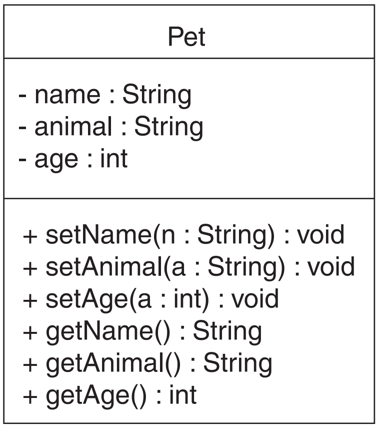
1. The constructor cannot have a return type, not even `void`.

3. The parentheses are missing. The statement should read:
`Rectangle box = new Rectangle();`

5. The `square` methods must have different parameter lists. Both accept an `int`.

### Algorithm Workbench

1. a. UML diagram:


Pet
- name : String
- animal : String
- age : int
+ setName(n : String) : void
+ setAnimal(a : String) : void
+ setAge(a : int) : void
+ getName() : String
+ getAnimal() : String
+ getAge() : int

b. Class code:

```
public class Pet
{
   private String name;      // The pet's name
   private String animal;    // The type of animal
   private int age;          // The pet's age
   /**
      setName method
      @param n The pet's name.
   */
   public void setName(String n)
   {
      name = n;
   }
   /**
      setAnimal method
      @param a The type of animal.
   */
   public void setAnimal(String a)
   {
      animal = a;
   }
   /**
      setAge method
      @param a The pet's age.
   */
   public void setAge(int a)
   {
      age = a;
   }
   /**
      getName method
      @return The pet's name.
   */
   public String getName()
   {
      return name;
   }
   /**
      getAnimal method
      @return The type of animal.
   */
   public String getAnimal()
   {
      return animal;
   }
   /**
      getAge method
      @return The pet's age.
   */
   public int getAge()
   {
      return age;
   }
}
```

3. a. 
```
public Square()
{
   sideLength = 0.0;
}
```
b. 
```
public Square(double s)
{
   sideLength = s;
}
```

## Short Answer

1. A class is a collection of programming statements that specify the attributes and methods that a particular type of object may have. You should think of a class as a "blueprint" that describes an object. An instance of a class is an actual object that exists in memory.

3. An accessor method is a method that gets a value from a class's field but does not change it. A mutator method is a method that stores a value in a field or in some other way changes the value of a field.

5. Methods that are members of the same class.

7. It looks in the current folder or directory for the file `Customer.class`. If that file does not exist, the compiler searches for the file `Customer.java` and compiles it. This creates the file `Customer.class`, which makes the `Customer` class available. The same procedure is followed when the compiler searches for the `Account` class.

9. If you do not write a constructor for a class, Java automatically provides one.

11. By their signatures, which include the method name and the data types of the method parameters, in the order that they appear.

# Chapter 7

## Multiple Choice and True/False

1. b
3. b
5. c
7. b
9. d
11. c
13. a
15. False
17. True
19. True
21. False
23. True

## Find the Error

1. The size declarator cannot be negative.
3. The loop uses the values 1 through 10 as subscripts. It should use 0 through 9.
5. A subscript should be used with words, such as `words[0].toUpperCase()`.

## Algorithm Workbench

1.
```
for (int i = 0; i < 20; i++)
   System.out.println(names[i]);
```

3.
```
a. String[] scientists = {"Einstein", "Newton",
                          "Copernicus", "Kepler"};
b. for (int i = 0; i < scientists.length; i++)
      System.out.println(scientists[i]);
c. int total = 0;
   for (int i = 0; i < scientists.length; i++)
      total += scientists[i].length();
   System.out.println("The total length is " + total);
```

5.
```
a. // Define the arrays.
   int[] id = new int[10];
   double[] weeklyPay = new double[10];
b. // Display each employee's gross weekly pay.
   for (int i = 0; i < 10; i++)
   {
      System.out.println("The pay for employee " +
                         id[i] + " is $" + weeklyPay[i]);
   }
```

7.
```
final int NUM_ROWS = 30;
final int NUM_COLS = 10;
int total = 0;
double average;
for (int row = 0; row < grades.length; row++)
{
   for (int col = 0; col < grades[row].length; col++)
   {
       total += grades[row][col];
   }
}
average = (double) total / (NUM_ROWS * NUM_COLS);
```

9.
```
double total = 0.0; // Accumulator
// Sum the values in the array.
for (int row = 0; row < 10; row++)
{
   for (int col = 0; col < 20; col++)
      total += values[row][col];
}
```

11.
```
// Create an ArrayList.
ArrayList<String> cars = new ArrayList<String>();
// Add three car names to the ArrayList.
cars.add("Porsche");
cars.add("BMW");
cars.add("Jaguar");
// Display the contents of cars.
for (String str : cars)
   System.out.println(str);
```

## Short Answer

1. The size declarator is used in a definition of an array to indicate the number of elements the array will have. A subscript is used to access a specific element in an array.

3. a. 2
   b. 14
   c. 8

5. Because this statement merely makes `array1` reference the same array that `array2` references. Both variables will reference the same array. To copy the contents of `array2` to `array1`, the contents of `array2`'s individual elements will have to be assigned to the elements of `array1`.

7. It will have to read all 10,000 elements to find the value stored in the last element.

# Chapter 8

## Multiple Choice and True/False

1. c
3. a
5. b
7. d
9. c
11. b
13. False
15. False
17. True

## Find the Error

1. The static method `setValues` cannot refer to the non-static fields `x` and `y`.

## Algorithm Workbench

1. a.
```
public String toString()
{
   String str;
   str = "Radius: " + radius + " Area: " + getArea();
   return str;
}
```
b.
```
public boolean equals(Circle c)
{
   boolean status;
   if (c.getRadius() == radius)
      status = true;
   else
      status = false;
   return status;
}
```
c.
```
public boolean greaterThan(Circle c)
{
   boolean status;
   if (c.getArea() > getArea())
      status = true;
   else
      status = false;
   return status;
}
```
3.
```
enum Pet { DOG, CAT, BIRD, HAMSTER }
```

### Short Answer

1. Access a non-static member.
3. When a variable is passed as an argument, a copy of the variable's contents is passed. The receiving method does not have access to the variable itself. When an object is passed as an argument, a reference to the object (which is the object's address) is passed. This allows the receiving method to have access to the object.
5. It means that an aggregate relationship exists. When an object of class B is a member of class A, it can be said that class A "has a" class B object.
7. It is not advisable because it will allow access to the private fields. The exception to this is when the field is a `String` object. This is because `String` objects are immutable, meaning that they cannot be changed.
9. a) `Color`

   b) `Color.RED, Color.ORANGE, Color.GREEN, Color.BLUE`

   c) `Color myColor = Color.BLUE;`
11. When there are no references to it.

## Chapter 9

### Multiple Choice and True/False

1. c
3. a
5. a
7. b
9. d
11. a
13. d
15. False
17. False
19. True
21. True
23. False

### Find the Error

1. The `valueOf` method is static. It should be called like this:
   `str = String.valueOf(number);`
3. The very first character is at position 0, so the statement should read:
   `str.setCharAt(0, 'Z');`

### Algorithm Workbench

1. `if (Character.toUpperCase(choice) == 'Y')`

   or

   `if (Character.toLowerCase(choice) == 'y')`

3.
```
int total = 0;
for (int i = 0; i < str.length(); i++)
{
   if (Character.isDigit(str.charAt(i)))
      total++;
}
```

5.
```
public static boolean dotCom(String str)
{
   boolean status;
   if (str.endsWith(".com"))
      status = true;
   else
      status = false;
   return status;
}
```

7.
```
public static void upperT(StringBuilder str)
{
   for (int i = 0; i < str.length(); i++)
   {
      if (str.charAt(i) == 't')
         str.setCharAt(i, 'T');
   }
}
```

9.
```
if (d <= Integer.MAX_VALUE)
   i = (int) d;
```

### Short Answer

1. This will improve the program's efficiency by reducing the number of `String` objects that must be created and then removed by the garbage collector.
3. Converts a number to a string.

## Chapter 10

### Multiple Choice and True/False

1. b
3. d
5. a

7. c
9. a
11. a
13. c
15. c
17. c
19. d
21. c
23. True
25. False
27. False
29. True
31. True
33. True

## Find the Error

1. The `Car` class header should use the word `extends` instead of `expands`.
3. Because the `Vehicle` class does not have a default constructor or a no-arg constructor, the `Car` class constructor must call the `Vehicle` class constructor.

## Algorithm Workbench

1. `public class Poodle extends Dog`
3.

```
public abstract class B
{
   private int m;
   protected int n;

   public void setM(int value)
   {
      m = value;
   }
   public void setN(int value)
   {
      n = value;
   }
   public int getM()
   {
      return m;
   }
   public int getN()
   {
      return n;
   }
   public abstract double calc();
}

public class D extends B
{
   private double q;
   protected double r;
   public void setQ(double value)
   {
      q = value;
   }
   public void setR(double value)
   {
      r = value;
   }
   public double getQ()
   {
      return q;
   }
   public double getR()
   {
      return r;
   }
   public double calc()
   {
      return q * r;
   }
}
```

5. `setValue(10);`

   or

   `super.setValue(10);`

7.
   ```
   public class Stereo extends SoundSystem
                       implements CDPlayable,
   ```

```
                    TunerPlayable,
                    CassettePlayable
```

9. `Computable half = x -> x / 2;`

### Short Answer

1. When an “is a” relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special.
3. `Dog` is the superclass and `Pet` is the subclass.
5. No.
7. Overloading is when a method has the same name as one or more other methods, but a different parameter list. Although overloaded methods have the same name, they have different signatures. When a method overrides another method, however, they both have the same signature.
9. At runtime.
11. An abstract class is not instantiated itself, but serves as a superclass for other classes. The abstract class represents the generic or abstract form of all the classes that inherit from it.
13. The class must implement an interface, or extend a superclass.
15. An expression that creates an object that implements a functional interface.

## Chapter 11

### Multiple Choice and True/False

1. b
3. a
5. b
7. d
9. c
11. c
13. d
15. c
17. True
19. False
21. True
23. False

## Find the Error

1. The `try` block must appear first.
3. The `catch (Exception e)` statement and its block should appear after the other catch blocks, because this is a more general exception than the others.

## Algorithm Workbench

1. `B`
   `D`

3.
```
public static int arraySearch(int[] array, int value)
                              throws Exception
{
   int i;            // Loop control variable
   int element;      // Element the value is found at
   boolean found;    // Flag indicating search results

   // Element 0 is the starting point of the search.
   i = 0;

   // Store the default values element and found.
   element = -1;
   found = false;

   // Search the array.
   while (!found && i < array.length)
   {
      if (array[i] == value)
      {
         found = true;
         element = i;
      }
      i++;
   }
   if (element == -1)
      throw new Exception("Element not found.");
   else
      return element;
}
```

5.
```
public class NegativeNumber extends Exception
{
   /**
      No-arg constructor
   */

   public NegativeNumber()
   {
      super("Error: Negative number");
   }

   /**
      The following constructor accepts the number that
      caused the exception.
      @param n The number.
   */
   public NegativeNumber(int n)
   {
      super("Error: Negative number: " + n);
   }
}
```

7. `public int getValueFromFile() throws IOException, FileNotFoundException`

9. 
```
FileOutputStream fstream =
                    new FileOutputStream("Configuration.dat");
```

11. 
```
FileOutputStream outStream =
                     new FileOutputStream("ObjectData.dat");
ObjectOutputStream objectOutputFile =
                     new ObjectOutputStream(outStream);
objectOutputFile.writeObject(r);
```

## Short Answer

1. An exception object has been created in response to an error that has occurred.
3. Control of the program is passed to the previous method in the call stack (that is, the method that called the offending method). If that method cannot handle the exception, then control is passed again, up the call stack, to the previous method. This continues until control reaches the `main` method. If the `main` method does not handle the exception, then the program is halted and the default exception handler handles the exception.
5. The first statement after that `try/catch` construct.
7. Any object that inherits from the `Throwable` class.
9. Unchecked exceptions are those that inherit from the `Error` class or the `RuntimeException` class. You should not handle these exceptions because the conditions that cause them can rarely be dealt with within the program. All of the remaining exceptions (that is, those that do not inherit from `Error` or `RuntimeException`) are *checked* exceptions. These are the exceptions that you should handle in your program.
11. All of the data stored in a text file is formatted as text. Even numeric data is converted to text when it is stored in a text file. You can view the data stored in a text file by opening it with a text editor, such as Notepad. In a binary file, data is not formatted as text. Subsequently, you cannot view a binary file's contents with a text editor.
13. When an object is serialized it is converted to a series of bytes, which are written to a file. When the object is deserialized, the series of bytes are converted back into an object.

## Chapter 12

### Multiple Choice and True/False

1. b
3. a
5. c
7. a
9. b
11. a
13. b
15. False
17. True

### Find the Error

1. The parameter for the overridden `start` method must be a `Stage` object, not a `Scene` object.
3. The argument that is passed to the `setAlignment` method should be `Pos.CENTER`.
5. The correct code is:

```
myImageView.setFitWidth(100);
myImageView.setFitHeight(100);
```

### Algorithm Workbench

1. `HBox hbox = new HBox(label1, label2, label3);`
3. `primaryStage.setScene(scene);`
5.

```
Image image = new Image("file:Cat.png");
ImageView imageView = new ImageView(image);
```

7. `HBox hbox = new HBox(10, label1, label2, label3);`
9. `gridPane.add(button, 2, 5);`
11.

```
class ButtonClickHandler implements EventHandler<ActionEvent>
{
    @Override
    void handle(ActionEvent event)
    {
        outputLabel.setText("Hello World");
    }
}
```

13.
```
myButton.setOnAction(event ->
{
   outputLabel.setText("Hello World");
};
```

### Short Answer

1. A program that responds to the user's actions.
3. What is the purpose of the `Application` class's abstract `start` method? It is the entry point for a JavaFX application.
5. `javafx.scene.control`
7. The aspect ratio is the ratio of the image's width to the image's height. If you change an image's aspect ratio, the image will appear stretched (either horizontally or vertically). The `ImageView` class has a method named `setPreserveRatio` that you can call to make sure that an image's aspect ratio is preserved.
9. You call the control's `getText` method. The method returns the value that has been entered into the TextField as a `String`.
11. BorderPane

## Chapter 13

### Multiple Choice and True/False

1. b
3. d
5. a
7. a
9. c
11. c
13. a
15. d
17. True
19. True
21. False
23. True

### Find the Error

1. The = operator should be a colon.
3. The last two statements should be:

```
radio1.setToggleGroup(radioGroup);
radio2.setToggleGroup(radioGroup);
```

## Algorithm Workbench

1.
```
.label {
   -fx-font-size: 24pt;
   -fx-font-style: italic;
   -fx-font-weight: bold;
}
```

3.
```
.label, .button, .text-field {
   -fx-font-size: 24pt;
   -fx-font-style: italic;
   -fx-font-weight: bold;
}
```

5. `rgb(100, 20, 255);`

7. `calcButton.setId("calc-button");`

9.
```
radio1.setToggleGroup(radioGroup);
radio2.setToggleGroup(radioGroup);
radio3.setToggleGroup(radioGroup);
```

11.
```
ListView<String> snackListView = new ListView<>();
snackListView.getItems().addAll("cheese", "olives", "crackers");
```

13.
```
Slider slider = new Slider(0.0, 1000.0, 500.0);
slider.setOrientation(Orientation.HORIZONTAL);
slider.setShowTickMarks(true);
slider.setMajorTickUnit(100);
slider.setMinorTickCount(25);
slider.setShowTickLabels(true);
```

## Short Answer

1. A type selector is used to apply a style definition to all nodes of a specific type. An ID selector can be used to apply a style definition to a node with a specific ID.

3. `RadioButtons`

5. You group `RadioButtons` together with a `ToggleGroup` so only one of the `RadioButtons` can be selected at a time.

7. An editable `ComboBox` allows the user to select an item from a drop-down list, or type input into a field that is similar to a `TextField`. By default, `ComboBox` controls are uneditable, which means that the user cannot type input into the control; he or she can only select items from a dropdown list.

9. `ComboBox`

# Chapter 14

## Multiple Choice and True/False

1. d
3. a
5. c
7. a
9. d
11. True
13. True
15. False

## Algorithm Workbench

1. `Line myLine = new Line(50, 25, 150, 125);`

3. `Rectangle myRectangle = new Rectangle(100, 200, 175, 150);`

5.
```
Polygon poly = new Polygon(100.0, 100.0,
                           200.0, 100.0,
                           200.0, 200.0,
                           100.0, 200.0);
```

7. `myBox.setRotate(45.0);`

9.
```
Circle myCircle = new Circle(0, 100, 30);
TranslateTransition ttrans =
         new TranslateTransition(new Duration(5000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(100);
ttrans.setToX(200);
ttrans.setToY(150);
ttrans.play();
```

## Short Answer

1. (639, 0)
3. `setFromX` and `setFromY`
5. `setFromAngle`
7. `setByAngle`

9. `setToX` and `setToY`
11. `BoxBlur`, `GuassianBlur`, and `MotionBlur`

## Chapter 15

### Multiple Choice and True/False

1. b
3. d
5. d
7. True
9. False

### Find the Error

1. The recursive method, `myMethod`, has no base case. So, it has no way of stopping.

### Algorithm Workbench

1.

```
public static void main(String [] args)
{
   String str = "test string";
   display(str, 0);
}
public static void display(String str, int pos)
{
   if (pos < str.length())
   {
      System.out.print(str.charAt(pos));
      display(str, pos + 1);
   }
}
```

3. `10`
5. `55`
7.

```
public static int factorial(int num)
{
   int fac = 1;
   for (int i = 1; i <=num; i++)
   {
      fac = fac * i;
   }
   return fac;
}
```

### Short Answer

1. An iterative algorithm uses a loop to solve the problem, while a recursive algorithm uses a method that calls itself.
3. For Question 3 the base case is reached when `arg` is equal to 10. For Question 4 the base case is also reached when `arg` is equal to 10. For Question 5 the base case is reached when `num` is less than or equal to 0.
5. Recursive algorithms are usually less efficient than iterative algorithms. This is because a method call requires several actions to be performed by the JVM, such as allocating memory for parameters and local variables, and storing the address of the program location where control returns after the method terminates.
7. The value of an argument is usually reduced.

## Chapter 16

### Multiple Choice and True/False

1. a
3. a
5. c
7. a
9. b
11. True
13. True

### Find the Error

1. The numbers in the array are not sorted.

### Algorithm Workbench

1.

| Array Size | 50 Elements | 500 Elements | 10,000 Elements | 100,000 Elements | 10,000,000 Elements |
|---|---|---|---|---|---|
| Sequential Search (Average Comparisons) | 25 | 250 | 5,000 | 50,000 | 5,000,000 |
| Sequential Search (Maximum Comparisons) | 50 | 500 | 10,000 | 100,000 | 10,000,000 |
| Binary Search (Maximum Comparisons) | 6 | 9 | 14 | 17 | 24 |

### Short Answer

1. It will have to read all 10,000 elements of the array.
3. One time.

5. Because it moves the items in the array only by one element at a time.
7. 
   - Move 1 to element 0. The array now looks like this:

     1, 4, 3, 2
   - Move 2 to element 1. The array now looks like this:

     1, 2, 3, 4
   - The values are now in order. No further swaps are made.

## Chapter 17

### Multiple Choice and True/False

1. b
3. a
5. a, b, and c
7. a
9. d
11. False
13. False
15. True
17. False

### Find the Error

1. The type argument must also be provided in the variable declaration. The statement should read:
   ```
   ArrayList<String> myList = new ArrayList<String>();
   ```
3. The static method cannot refer to the class's type parameter `T`. It must define its own type parameter.

### Algorithm Workbench

1. 
   ```
   ArrayList<Customer> customers = new ArrayList<Customer>();
   ```
3. 
   ```
   public class MyType<T extends String>
   ```
5. 
   ```
   public static <T extends Comparable> T max(T a, T b)
   {
     if (a.compareTo(b) > 0)
        return a;
     else
        return b;
   }
   ```
7. 
   ```
   public class MyType<T extends Number, S extends String>
   ```

### Short Answer

1. At compile time.
3. When the Java compiler encounters generic code, it uses a process known as erasure to compile the code. The process is called erasure because the compiler erases the generic notation and substitutes an actual type for each type parameter.
5. `Object`.

## Chapter 18

### Multiple Choice and True/False

1. a
3. b
5. b
7. a
9. c
11. a
13. d
15. True
17. False
19. True
21. False

### Find the Error

1. The `LinkedHashSet` class does not implement the `List` interface. Replace with `LinkedList` or `ArrayList`.
3. You cannot use a `ListIterator` from an object that implements the `Set` interface.

### Algorithm Workbench

1. `List<String> sList = new ArrayList<>();`
3. 
```
while (it.hasNext())
{
   System.out.println(it.next());
}
```

### Short Answer

1. A source
3. You can change the class that implements the interface without changing the rest of the code.
5. By calling the `iterator()` method. Or if you want to, you can call the `listIterator()` method.
7. By internally generating code that uses an iterator.
9. Because `IntStream` allows streams of the primitive type `int`. You cannot get a stream of primitive types from `Stream<Integer>`.
11. Yes.
13. A map associates keys with values. A `HashMap` object will use the hash codes of the keys to find the keys and then from there, to find the values.

## Chapter 19

### Multiple Choice and True/False

1. b
3. c
5. a
7. d
9. c
11. True
13. False
15. True
17. False
19. False

### Find the Error

1. Replace `ref++` with `ref = ref.next;`
3. You need to set the `prev` reference of the second node. Add the code:

```
if (myList.next != null)
   myList.next.prev = myList;
```

5. You need to check to see if the list is now empty, and you need to set the `next` reference of the last node to null.

```
if (last == null)
   first = null;
else
   last.next = null;
```

### Algorithm Workbench

1.
```
static void print(Node ref)
{
   if (ref != null)
   {
      // Print the first element
      System.out.print(ref.value +  " ");
      // Print elements in the tail
      print(ref.next);
   }
}
```

3.
```
Node reverse(Node list)
{
   // Check base case
   if (list == null || list.next == null)
      return null;
   // reverse the tail
   Node revTail = reverse(list.next);
   // Put head at the end of the reversed tail
   list.next = null;
   Node last = revTail;
   while (last.next != null)
      last = last.next;
   last.next = list;

   // revTail is now head of the reversed list
   return revTail;
}
```

### Short Answer

1. They need to be passed a `Node` reference as parameter. `Node` is an implementation detail that should not be made public.
3. It is the zero-based position of the element in the list.
5. Successive elements of the list are not necessarily stored in consecutive memory locations. Instead, the list stores with each element a reference to its successor.

## Chapter 20

### Multiple Choice and True/False

1. b
3. b
5. c

7. False
9. False

### Find the Error

1. The flow of execution never reaches the statement `top --;`
3. `pop` forgets to remove the value from the stack.
5. `dequeue` should remove from the front, not the rear. Also, the index should wrap to the beginning of the array when it gets to the end.

### Algorithm Workbench

1.

```
static boolean palindrome(String s)
{
   Stack<Character> stack = new Stack<Character>();
   // Push a copy of the string onto the stack
   for(int k = 0; k < s.length(); k++)
      stack.push(s.charAt(k));
   // Compare the two copies using the stack
   for (int k = 0; k < s.length(); k++)
   {
     if (s.charAt(k) != stack.pop())
        return false;
   }
   // No discrepancy found, so s is  a palindrome
   return true;
}
```

3. To add *x* to the queue, push *x* onto the first stack. To remove an item from the queue, pop everything from the first stack and store it into the second stack, then pop the item at the top of the second stack and save it to be returned. Pop everything from the second stack back onto the first stack. Return the saved item.

### Short Answer

1. A stack
3. A stack
5. To facilitate garbage collection.

## Chapter 21

### Multiple Choice and True/False

1. d
3. c

5. c
7. a
9. b
11. False
13. True
15. True
17. True
19. True
21. False

### Find the Error

1. The function does not use the values returned from the recursive calls. Should be:

```
if (tree!= null)
   return NodeCount(tree.left)+NodeCount(tree.right)+1;
else
   return 0;
```

3. The code will fail if `tree` is null.

### Short Answer

1. It keeps the height of the tree shallow, making algorithms that traverse a path of the tree from the root to a leaf more efficient.
3. The length of the longest path from the root to a leaf.
5. A collection that stores elements from an ordered set, and from which elements are removed according to that order, with the least element in the order being removed first.
7. The AVL tree must satisfy a balance condition: the difference in the heights of the two subtrees of any node must be at most 1.

### Algorithm Workbench

1.

```
Node preorder(Node bTree)
{
   if (tree != null)
   {
       System.out.print(bTree.value + " ");
       preorder(bTree.left);
       preorder(bTree.right);
   }
}
```

3.
```
int treeSize(Node bTree)
{
   if (bTree == null)
      return 0;
   else
      return treeSize(bTree.left) + treeSize(bTree.right) + 1;
}
```

5.
```
boolean contains(Node btree, int x)
{
   if (btree == null)
      return false;
   if (btree.value == x)
      return true;
   if (contains(btree.left, x))
      return true;
   if (contains(btree.right, x))
      return true;
   else
      return false;
}
```

7.
```
static class DecOrder implements Comparator<String>
{
   public int compare(String x, String y)
   {
      if (x.length() == y.length())
         return x.compareTo(y);
      else
         return x.length()-y.length();
   }
}
```

## **Chapter 22** (Available on the Book's Companion Website)

### Multiple Choice and True/False

1. b
3. a
5. d
7. b
9. a
11. c
13. b
15. d

17. c
19. True
21. False
23. False

### Find the Error

1. The string "French Roast Dark" should be enclosed in single quotes instead of double quotes.
3. The last statement should call the `executeQuery` method instead of the `execute` method.

### Algorithm Workbench

1.
   - `INTEGER or INT`
   - `REAL`
   - `CHARACTER(n) or CHAR(n) or VARCHAR(n)`
   - `DOUBLE`

3.
```
SELECT * FROM Stock
```

5.
```
SELECT TradingSymbol, NumShares FROM Stock
```

7.
```
SELECT * FROM Stock
WHERE TradingSymbol LIKE 'SU%'
```

9.
```
SELECT TradingSymbol, NumShares FROM Stock
WHERE SellingPrice > PurchasePrice AND
NumShares > 100
ORDER BY NumShares
```

11.
```
UPDATE Stock
SET TradingSymbol = 'ABC'
WHERE TradingSymbol = 'XYZ'
```

13.
```
Connection conn = DriverManager.getConnection(DB_URL);
```

15.
```
ResultSet result = stmt.executeQuery(sqlStatement);
```

17.
```
CREATE TABLE Car
( Manufacturer CHAR(25),
  Year INTEGER,
  VehicleID CHAR(20) )
```

### Short Answer

1. Traditional text and binary files are not practical when a large amount of data must be stored and manipulated. Many businesses keep hundreds of thousands, or even millions, of data items in files. When a text or binary file contains this much data, simple operations such as searching, inserting, and deleting become cumbersome and inefficient.

3. The data that is stored in a database is organized into one or more tables. Each table holds a collection of related data. The data that is stored in a table is then organized into rows and columns. A row is a complete set of information about a single item. The data that is stored in a row is divided into columns. Each column is an individual piece of information about the item.
5. A result set is an object that is somewhat similar to a collection, and contains the results of an SQL statement.
7. The first row is row 1. The first column is column 1.
9. A foreign key is a column in one table that references a primary key in another table.

## **Chapter 23** (Available on the Book's Companion Website)

### Multiple Choice and True/False

1. c
3. c
5. d
7. b
9. b
11. a
13. False
15. True
17. False
19. False

### Find the Error

1. The `x` is missing in the package name. The statement should read:
   `import javax.swing.*;`
3. The arguments passed to the `GridLayout` constructor are reversed. This statement creates 10 rows and 5 columns.
5. You do not create an instance of `BorderFactory`. Instead you call one of its static methods to create a `Border` object. The statement should read:
   `panel.setBorder(BorderFactory.createTitledBorder("Choices"));`

### Algorithm Workbench

1. `myWindow.setSize(500, 250);`
3. `myWindow.setVisible(true);`

5. `setLayout(new FlowLayout(FlowLayout.LEFT));`
7. `panel.add(button, BorderLayout.WEST);`
9. `panel.setBorder(BorderFactory.createLineBorder(Color.blue, 2));`

### Short Answer

1. The window is hidden from view, but the application does not end.
3. Radio buttons are normally used to select one of several possible items. Because a mutually exclusive relationship usually exists between radio buttons, only one of the items may be selected. Check boxes, which may appear alone or in groups, allow the user to make yes/no or on/off selections. Because there is not usually a mutually exclusive relationship between check boxes, the user can select any number of them when they appear in a group.

## **Chapter 24** (Available on the Book's Companion Website)

### Multiple Choice and True/False

1. d
3. b
5. a
7. c
9. b
11. c
13. a
15. b
17. a
19. b
21. a
23. False
25. False
27. True
29. True
31. True
33. False
35. True

## Find the Error

1. The argument `false` should have been passed to the `setEditable` method.
3. You should pass `list` as an argument to the `JScrollPane` constructor:
   ```
   JScrollPane scrollPane = new JScrollPane(list);
   ```
5. The second statement should read:
   ```
   label.setIcon(image);
   ```
7. The statement should read:
   ```
   JTextArea textArea = new JTextArea (5, 20);
   ```

## Algorithm Workbench

1. 
   ```
   JTextField textField = new JTextField(20);

   textField.setEditable(false);
   ```
3. 
   ```
   dayList.setVisibleRowCount(4);

   JScrollPane scrollPane = new JScrollPane(dayList);
   ```
5. 
   ```
   selectionIndex = myComboBox.getSelectedIndex();
   ```
7. 
   ```
   ImageIcon image = new ImageIcon("picture.gif");

   label.setIcon(image);
   ```
9. 
   ```
   JFileChooser fileChooser = new JFileChooser();

   int status = fileChooser.showOpenDialog(null);
   if (status == JFileChooser.APPROVE_OPTION)
   {
      File selectedFile = fileChooser.getSelectedFile();
      String filename = selectedFile.getPath();
   }
   ```
11. 
   ```
   // Create an Open menu item.
   JMenuItem openItem = new JMenuItem("Open");
   openItem.setMnemonic(KeyEvent.VK_O);
   openItem.addActionListener(new OpenListener());
   // Create a Print menu item.
   JMenuItem printItem = new JMenuItem("Print");
   printItem.setMnemonic(KeyEvent.VK_P);
   printItem.addActionListener(new PrintListener());
   // Create an Exit menu item.
   JMenuItem exitItem = new JMenuItem("Exit");
   exitItem.setMnemonic(KeyEvent.VK_X);
   exitItem.addActionListener(new ExitListener());
   // Create a JMenu object for the File menu.
   JMenu fileMenu = new JMenu("File");
   fileMenu.setMnemonic(KeyEvent.VK_F);
   // Add the menu items to the File menu.
   fileMenu.add(openItem);
   ```

```
fileMenu.add(printItem);
fileMenu.add(exitItem);
// Create the menu bar.
JMenuBar menuBar = new JMenuBar();
// Add the file menu to the menu bar.
menuBar.add(fileMenu);
// Set the window's menu bar.
setJMenuBar(menuBar);
```

### Short Answer

1. Single selection mode
3. An uneditable combo box combines a button with a list, and allows the user to select only items from its list. An editable combo box combines a text field and a list. In addition to selecting items from the list, the user may also type input into the text field. The default type of combo box is uneditable.
5. A mnemonic is a key on the keyboard that you press in combination with the Alt key to quickly access a component such as a button. When you assign a mnemonic to a button, the user can click the button by holding down the a key and pressing the mnemonic key.
7. A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component. The box usually gives a short description of what the component does.
9. The item is deselected, which causes the check mark to disappear. The checked menu item component also generates an action event.
11. Because, as the user moves the `JSlider` component's knob, it will only take on values within its established range.

## **Chapter 25** (Available on the Book's Companion Website)

### Multiple Choice and True/False

1. c
3. b
5. b
7. d
9. c
11. a
13. d
15. c
17. c

19. c
21. b
23. True
25. True
27. False
29. True
31. True
33. False
35. False

### Find the Error

1. The tag should specify the file `MyApplet`.class instead of `MyApplet.java`.
3. Call `repaint` instead of `paint`.
5. The class must provide all of the methods specified by the `MouseListener` interface.

### Algorithm Workbench

1. `<center><h1>My Home Page</h1></center>`
3. Line 1: Change `JFrame` to `JApplet`

   Line 3: Change to `public void init()`

   Line 5: Delete

   Line 6: Delete

   Line 8: Delete

   Line 9: Delete

   Line 15: Delete

   Line 16: Delete

   Line 17: Delete
5. 
```
private class MyMouseMotionListener extends MouseAdapter
{
   public void mouseMoved(MouseEvent e)
   {
      mouseMovements += 1;
   }
}
```

### Short Answer

1. It is executed by the user's system.
3. Applets are important because they can be used to extend the capabilities of a Web page. Web pages are normally written in Hypertext Markup Language (HTML). HTML is limited, however, because it merely describes the content and layout of a Web page, and creates links to other files and Web pages. HTML does not have sophisticated abilities such as performing math calculations and interacting with the user. A programmer can write a Java applet to perform these types of operations and associate it with a Web page.
5. Some browsers, such as older versions of Microsoft Internet Explorer, do not directly support the Swing classes in applets. These browsers require a plug-in in order to run applets that use Swing components. If you are writing an applet for other people to run on their computers, there is no guarantee that they will have the required plug-in. If this is the case, you should use the AWT classes instead of the Swing classes for the components in your applet.
7. When the component is first displayed and is called again any time the component needs to be redisplayed.
9. If you want to load the sound file and keep it in memory so it can be played more than once, or if you want to play the sound file repeatedly.

CASE STUDY

# 1

# Calculating Sales Commission

Hal's Home Computer Emporium is a retail seller of home computers. Hal's sales staff works strictly on commission. At the end of the month, each salesperson's commission is calculated according to Table CS1-1.

**Table CS1-1** Sales commission rates

| Sales This Month | Commission Rate |
|---|---|
| less than $10,000 | 5% |
| $10,000–14,999 | 10% |
| $15,000–17,999 | 12% |
| $18,000–21,999 | 15% |
| $22,000 or more | 16% |

For example, a salesperson with $16,000 in monthly sales will earn a 12% commission ($1,920). Another salesperson with $20,000 in monthly sales will earn a 15% commission ($3,000).

Because the staff gets paid once per month, Hal allows each employee to take up to $1,500 per month in advance. When sales commissions are calculated, the amount of each employee's advanced pay is subtracted from the commission. If any salesperson's commission is less than the amount of this advance, he or she must reimburse Hal for the difference.

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

Here are two examples: Beverly and John have $21,400 and $12,600 in sales, respectively. Beverly's commission is $3,210 and John's commission is $1,260. Both Beverly and John took $1,500 in advance pay. At the end of the month, Beverly gets a check for $1,710, but John must pay $240 back to Hal.

Now we will examine a program that eases the task of calculating the end-of-month commission.

## Program Design

The program must perform the following general steps:

1. Ask the user to enter the salesperson's monthly sales.
2. Ask the user to enter the amount of advanced pay.
3. Use the amount of monthly sales to determine the commission rate.
4. Calculate the commission.
5. Calculate the salesperson's pay by subtracting the amount of advanced pay from the commission. If the amount is negative, the salesperson must reimburse the company.

Table CS1-2 lists and describes the program's variables.

**Table CS1-2** Variables

| Variable | Description |
|---|---|
| `sales` | A `double` variable to hold a salesperson's total monthly sales. |
| `rate` | A `double` variable to hold the salesperson's commission rate. |
| `commission` | A `double` variable to hold the commission. |
| `advance` | A `double` variable to hold the amount of advanced pay the salesperson has drawn. |
| `Pay` | A `double` variable to hold the salesperson's amount of gross pay. |

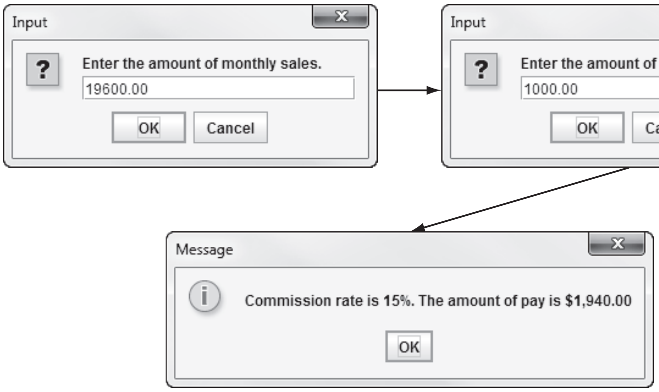
Programmers often write parts of a program in pseudocode before writing the actual Java code. Recall from Chapter 1 that *pseudocode* is a cross between human language and a programming language. It is an informal way of writing down each operation that a program must perform, and is especially helpful when designing an algorithm. Although the computer can't understand pseudocode, most programmers find pseudocode helpful in creating a model of a program. When the pseudocode model is complete, it can be converted to actual code. Here is a pseudocode model for the sales commission program:

```
Display "Enter the amount of monthly sales".
Read sales.
Display "Enter the amount of advanced pay".
Read advance.
If sales is less than 10,000
    Store 0.05 in rate.
else if sales is less than 15,000
    Store 0.1 in rate.
```

```
else if sales is less than 18,000
    Store 0.12 in rate.
else if sales is less than 22,000
    Store 0.15 in rate.
else
    Store 0.16 in rate.
commission = sales * rate.
pay = commission – advance.
Display pay.
```

Code Listing CS1-1 shows the actual program. Figure CS1-1 shows an example of interaction with the program.

**Figure CS1-1** Interaction with the `HalsCommission` program



Enter the amount of monthly sales.
19600.00

Enter the amount of advanced pay.
1000.00

Commission rate is 15%. The amount of pay is $1,940.00

**Code Listing CS1-1** (`HalsCommission.java`)

```
import javax.swing.JOptionPane;   // Needed for JOptionPane

/**
   This program calculates a salesperson's gross
   pay at Hal's Computer Emporium.
*/

public class HalsCommission
{
   public static void main(String[] args)
   {
      String input;       // To hold the user's input
      double sales;       // Monthly sales
      double rate;        // Commission rate
      double commission;  // Amount of commission
      double advance;     // Advanced pay
      double pay;         // Salesperson’s pay

      // Get the monthly sales.
      input = JOptionPane.showInputDialog(“Enter the amount ” +
                                          “of monthly sales.”);
      sales = Double.parseDouble(input);

      // Get the advanced pay.
      input = JOptionPane.showInputDialog(“Enter the amount ” +
                                          “of advanced pay.”);
      advance = Double.parseDouble(input);

      // Determine the rate of commission.
      if (sales < 10000)
         rate = 0.05;           // 5% commission rate
      else if (sales < 15000)
         rate = 0.1;            // 10% commission rate
      else if (sales < 18000)
         rate = 0.12;           // 12% commission rate
      else if (sales < 22000)
         rate = 0.15;           // 15% commission rate
      else
         rate = 0.16;           // 16% commission rate

      // Calculate the amount of commission.
      commission = rate * sales;

      // Calculate the salesperson’s pay.
      pay = commission - advance;

      // Display the salesperson’s commission rate and pay.
      JOptionPane.showMessageDialog(null,
         String.format(“Commission rate is %.2f. The amount of pay is $%,.2f”,
                       rate, pay));
      System.exit(0);
   }
}
```

CASE STUDY

# 2 The Amortization Class

The loan officer at one of the Central Mountain Credit Union's branch offices has asked you to write a loan amortization application to run on her desktop PC. The application should allow the user to enter the amount of a loan, the number of years of the loan, and the annual interest rate. An amortization report should then be saved to a text file.

## Calculations

The credit union uses the following formula to calculate the monthly payment of a loan:

$$Payment = \frac{Loan \times \dfrac{Rate}{12} \times Term}{Term - 1}$$

where:

$Loan$ = the amount of the loan,

$Rate$ = the annual interest rate, and

$Term = (1 + Rate/12)^{Years \times 12}$

## Report Requirements

The report produced by the program should show the monthly payment amount and the following information for each month in the loan period: amount applied to interest, amount applied to principal, and the balance. The following report may be used as a model. It shows all the required information on a one-year $5,000 loan at 5.9 percent annual interest.

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

```
Monthly Payment: $430.10
Month      Interest      Principal    Balance
-----------------------------------------------
  1        24.58         405.52       4,594.48
  2        22.59         407.51       4,186.97
  3        20.59         409.52       3,777.45
  4        18.57         411.53       3,365.92
  5        16.55         413.55       2,952.37
  6        14.52         415.59       2,536.78
  7        12.47         417.63       2,119.15
  8        10.42         419.68       1,699.47
  9        8.36          421.75       1,277.72
 10        6.28          423.82       853.90
 11        4.20          425.90       428.00
 12        2.10          428.00       0.00
```

The core of the program will be a class, `Amortization`, that holds the primary data, performs the mathematical calculations, and displays the report. Figure CS2-1 shows a UML diagram for the class.

Table CS2-1 lists and describes the class's fields.

**Figure CS2-1** UML diagram for the `Amortization` class

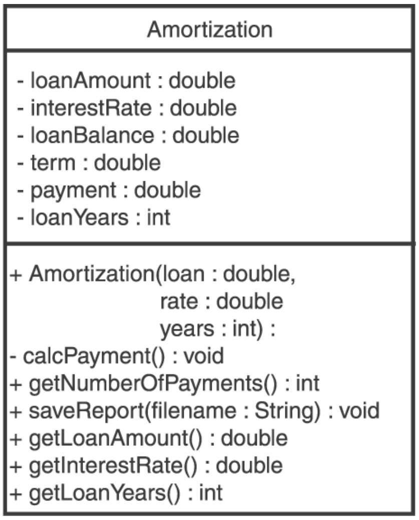

Amortization
- loanAmount : double
- interestRate : double
- loanBalance : double
- term : double
- payment : double
- loanYears : int
+ Amortization(loan : double,
rate : double
years : int) :
- calcPayment() : void
+ getNumberOfPayments() : int
+ saveReport(filename : String) : void
+ getLoanAmount() : double
+ getInterestRate() : double
+ getLoanYears() : int

**Table CS2-1** Amortization class fields

| Field | Description |
|---|---|
| `loanAmount` | A `double` variable to hold the amount of the loan. |
| `interestRate` | A `double` variable to hold the annual interest rate. |
| `loanBalance` | A `double` variable to hold the loan balance. |
| `term` | A `double` variable used in the calculation of the monthly payment. |
| `payment` | A `double` variable to hold the amount of the monthly payment. |
| `loanYears` | An `int` variable to hold the number of years of the loan. |

Table CS2-2 lists and describes the class's methods.

**Table CS2-2** Amortization class methods

| Method | Description |
|---|---|
| Constructor | The constructor accepts three arguments: the loan amount, the annual interest rate, and the number of years of the loan. These values are stored in their corresponding fields. The private method `calcPayment` is then called. |
| `calcPayment` | A private method that is used to calculate the monthly payment amount. The result is stored in the `payment` field. |
| `getNumberOfPayments` | Returns as an `int` the number of loan payments. |
| `saveReport` | Saves the amortization report to a text file. |
| `getLoanAmount` | Returns as a `double` the amount of the loan. |
| `getInterestRate` | Returns as a `double` the annual interest rate. |
| `getLoanYears` | Returns as an `int` the number of years of the loan. |

Code Listing CS2-1 shows the code for the class.

**Code Listing CS2-1** (Amortization.java)

```
import java.io.*;  // For file-related classes

/**
   This class stores loan information and creates a
   text file containing an amortization report.
*/
```

```

public class Amortization
{
   private double loanAmount;    // Loan Amount
   private double interestRate;  // Annual Interest Rate
   private double loanBalance;   // Monthly Balance
   private double term;          // Payment Term
   private double payment;       // Monthly Payment
   private int    loanYears;     // Years of Loan

   /**
      The constructor accepts the loan amount, the annual
      interest rate, and the number of years of the loan
      as arguments. The private method CalcPayment is then
      called.
      @param loan The loan amount.
      @param rate The annual interest rate.
      @param years The number of years of the loan.
   */

   public Amortization(double loan, double rate, int years)
   {
      loanAmount = loan;
      loanBalance = loan;
      interestRate = rate;
      loanYears = years;
      calcPayment();
   }

   /**
      The calcPayment method calculates the monthly payment
      amount. The result is stored in the payment field.
   */

   private void calcPayment()
   {
      // Calculate value of Term
      term =
        Math.pow((1+interestRate/12.0), 12.0 * loanYears);

      // Calculate monthly payment
      payment =
        (loanAmount * interestRate/12.0 * term) / (term - 1);
   }

   /**
      The getNumberOfPayments method returns the total number of
      payments to be made for the loan.
      @return The number of loan payments.
   */

   public int getNumberOfPayments()
   {
      return 12 * loanYears;
   }

   /**
      The saveReport method saves the amortization report to
      the file named by the argument.
      @param filename The name of the file to create.
   */

   public void saveReport(String filename) throws IOException
   {
      double monthlyInterest;  // The monthly interest rate
      double principal;        // The amount of principal
      //DecimalFormat dollar = new DecimalFormat(“#,##0.00”);
      FileWriter fwriter = new FileWriter(filename);
      PrintWriter outputFile = new PrintWriter(fwriter);

      // Print monthly payment amount.
      outputFile.println(String.format(
               “Monthly Payment: $%.2f”, payment));

      // Print the report header.
      outputFile.println(“Month\tInterest\tPrincipal\tBalance”);
      outputFile.println(“----------------------------------” +
                         “--------------”);

      // Display the amortization table.
      for (int month = 1; month <= getNumberOfPayments(); month++)
      {
        // Calculate monthly interest.
        monthlyInterest = interestRate / 12.0 * loanBalance;

        if (month != getNumberOfPayments())
        {
           // Calculate payment applied to principal
           principal = payment - monthlyInterest;
        }
        else   // This is the last month.
        {
           principal = loanBalance;
           payment = loanBalance + monthlyInterest;
        }

```

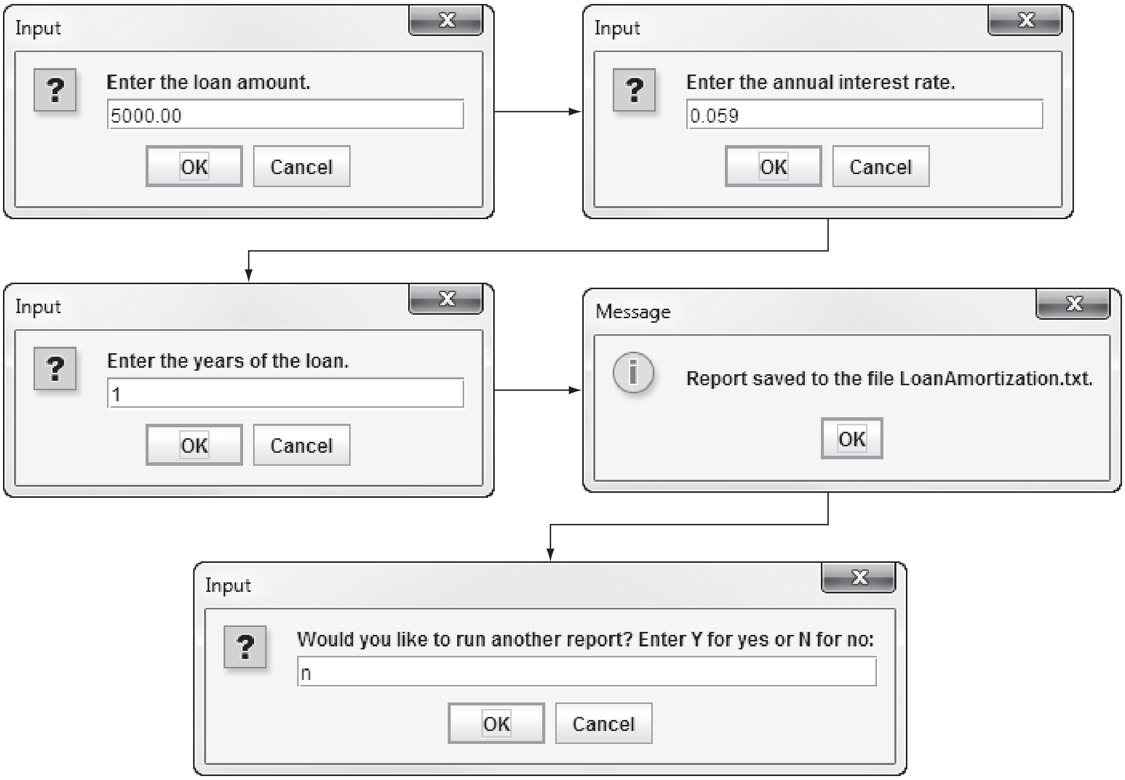
```
          // Calculate the new loan balance.
          loanBalance -= principal;

          // Display a line of data.
          outputFile.println(String.format(“%d\t%.2f\t\t%.2f\t\t%.2f”,
                             month, monthlyInterest, principal,
                             loanBalance));
      }

      // Close the file.
      outputFile.close();
   }

   /**
       The getLoanAmount method returns the loan amount.
       @return The value in the loanAmount field.
   */

   public double getLoanAmount()
   {
      return loanAmount;
   }

   /**
      The getInterestRate method returns the interest rate.
      @return The value in the interestRate field.
   */

   public double getInterestRate()
   {
      return interestRate;
   }

   /**
      The getLoanYears method returns the years of the loan.
      @return The value in the loanYears field.
   */

   public int getLoanYears()
   {
      return loanYears;
   }
}
```

## The Main Program

The main program code is shown in Code Listing CS2-2. First, it gets the amount of the loan, the annual interest rate, and the years of the loan as input from the user. It then creates an instance of the `Amortization` class and passes this data to the class’s constructor.

The program then saves the amortization report in the file `LoanAmortization.txt`. It asks the user whether he or she wants to run another report. If so, the program repeats these steps. Figure CS2-2 shows an example of interaction with the program.

**Figure CS2-2** Interaction with the `LoanReport` program



Enter the loan amount.
5000.00

Enter the annual interest rate.
0.059

Enter the years of the loan.
1

Report saved to the file LoanAmortization.txt.

Would you like to run another report? Enter Y for yes or N for no:
n

## Code Listing CS2-2 (`LoanReport.java`)

```java
import javax.swing.JOptionPane; // For the JOptionPane class

// The following import statement is required because the main
// method has a throws IOException clause. Although this program
// doesn't have code that directly performs file I/O, the import
// statement is still required because of IOException.
import java.io.*;

/**
   This program displays a loan amortization report.
*/

public class LoanReport
{
   public static void main(String[] args) throws IOException
   {
      String input;         // To hold user input
      double loan;          // Loan amount
      double interestRate;  // Annual interest rate
      int years;            // Years of the loan
      char again;           // To indicate if loop should repeat

      do
      {
         // Get the loan amount.
         input = JOptionPane.showInputDialog(“Enter the ” +
                                             “loan amount.”);
         loan = Double.parseDouble(input);

         // Validate the loan amount.
         // (No negative amounts.)
         while (loan < 0)
         {
            input = JOptionPane.showInputDialog(“Invalid amount. ” +
                                                “Enter the loan amount.”);
            loan = Double.parseDouble(input);
         }

         // Get the annual interest rate.
         input = JOptionPane.showInputDialog(“Enter the ” +
                                          “annual interest rate.”);
         interestRate = Double.parseDouble(input);
         // Validate the interest rate
         // (No negative amounts.)
         while (interestRate < 0)
         {
           input = JOptionPane.showInputDialog(“Invalid amount. ” +
                            “Enter the annual interest rate.”);
           interestRate = Double.parseDouble(input);
         }

         // Get the years of the loan.
         input = JOptionPane.showInputDialog(“Enter the ” +
                                        “years of the loan.”);
         years = Integer.parseInt(input);

         // Validate the number of years.
         // (No negative amounts.)
         while (years < 0)
         {
           input = JOptionPane.showInputDialog(“Invalid amount. ” +
                                 “Enter the years of the loan.”);
```

```
63              years = Integer.parseInt(input);
64            }
65
66            // Create and initialize an Amortization object.
67            Amortization am =
68                new Amortization(loan, interestRate, years);
69
70            // Save the report.
71            am.saveReport(“LoanAmortization.txt”);
72            JOptionPane.showMessageDialog(null, “Report saved to ” +
73                                  “the file LoanAmortization.txt.”);
74
75            // Do another report?
76            input = JOptionPane.showInputDialog(“Would you like ” +
77                             “to run another report? Enter Y for ” +
78                             “yes or N for no: ”);
79            again = input.charAt(0);
80
81         } while (again == ‘Y’ || again == ‘y’);
82
83         System.exit(0);
84      }
85   }
```

Contents of the file *LoanAmortization.txt*:

```
Monthly Payment: $430.10
Month       Interest     Principal     Balance
-----------------------------------------------
  1         24.58        405.52        4,594.48
  2         22.59        407.51        4,186.97
  3         20.59        409.52        3,777.45
  4         18.57        411.53        3,365.92
  5         16.55        413.55        2,952.37
  6         14.52        415.59        2,536.78
  7         12.47        417.63        2,119.15
  8         10.42        419.68        1,699.47
  9         8.36         421.75        1,277.72
 10         6.28         423.82        853.90
 11         4.20         425.90        428.00
 12         2.10         428.00        0.00
```

First, notice that a `do-while` loop (in lines 23 through 81) controls everything done in this program. Here is a condensed version of the loop:

```
23      do
24      {
```

*(Code to get the loan data and display the report.)*

```
75       // Do another report?
76       input = JOptionPane.showInputDialog("Would you like " +
77                        "to run another report? Enter Y for " +
78                        "yes or N for no: ");
79       again = input.charAt(0);
80
81    } while (again == 'Y' || again == 'y');
```

During each iteration, this loop first gathers the necessary loan data as input and then writes the amortization report. At the end of the iteration, the user is asked whether he or she wants to run another report. If the user enters `Y` or `y` for yes, the loop repeats. Otherwise, the loop terminates and the program ends.

Inside the loop, the code that gathers input from the user also validates the input. For example, here is the code in lines 25 through 37 that gets and validates the loan amount:

```
25       // Get the loan amount.
26       input = JOptionPane.showInputDialog("Enter the " +
27                                           "loan amount.");
28       loan = Double.parseDouble(input);
29
30       // Validate the loan amount.
31       // (No negative amounts.)
32       while (loan < 0)
33       {
34          input = JOptionPane.showInputDialog
35                 ("Invalid amount. Enter the loan amount.");
36          loan = Double.parseDouble(input);
37       }
```

If the user enters a negative number, the `while` loop displays an error message and asks the user to enter the loan amount again. The code that gets the annual interest rate and the years of the loan, in lines 39 through 64, also performs similar input validation. Once these values have been correctly entered, an instance of the `Amortization` class is created in lines 67 and 68:

```
Amortization am =
   new Amortization(loan, interestRate, years);
```

Then, the `saveReport method` is called in line 71 to save the amortization report to the file `LoanAmortization.txt`:

```
am.printReport("LoanAmortization.txt");
```

**NOTE:** Notice that Code Listing CS2-1 has the following import statement in line 1:

```
import java.io.*;
```

Although this program does not have any code that directly performs file I/O, we still have to have this `import` statement because of the `throws IOException` clause in the `main` method header. The `main` method has to have the `throws IOException` clause because it calls the `saveReport` method in the `Amortization` class.

CASE STUDY

# 3 The PinTester Class

The National Commerce Bank has hired you as a contract programmer. Your first assignment is to write a class that will be used by the bank's automated teller machines (ATMs) to validate a customer's personal identification number (PIN). Your class, which will be named `PinTester`, will be incorporated into a larger program that asks the customer to input his or her PIN on the ATM's numeric keypad. (PINs are multi-digit numbers. The program stores each digit in an element of an `int` array.) The program also retrieves a copy of the customer's actual PIN from a database, as an `int` array. If these arrays match, then the customer's identity is validated.

Table CS3-1 lists the fields that the class should have.

**Table CS3-1** `PinTester` class fields

| Field Name | Description |
|---|---|
| `userPIN` | This field should be an int array reference variable. It will be used to reference an array containing the PIN digits entered by the user. |
| `validPIN` | This field should be an int array reference variable. It will be used to reference an array containing the valid PIN digits retrieved from the database. |

Table CS3-2 lists the methods that the class should have.

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

**Table CS3-2** PinTester class methods

| Method | Description |
|---|---|
| Constructor | The constructor is to accept references to two int arrays as arguments. The array referenced by the first argument will contain the numbers entered by the customer. The array referenced by the second argument will contain the numbers retrieved from the bank's database. The constructor should create two arrays and reference them by the `userPIN` and `validPIN` fields. Then, it should copy the contents of the arrays passed as arguments to the `userPIN` and `validPIN` arrays. |
| `isValid` | This `boolean` function should return `true` if the `userPIN` and `validPIN` arrays are identical. Otherwise, it should return `false`. |

Code Listing CS3-1 shows the `PinTester` class.

**Code Listing CS3-1** (PinTester.java)

```
/**
   The PinTester class is used to compare the PIN that
   was entered by a user to the valid PIN.
*/

public class PinTester
{
   private int[] userPIN; // The PIN entered
   private int[] validPIN; // The valid PIN

   /**
      Constructor
      @param user The PIN digits entered by the user.
      @param valid The valid PIN digits.
   */

   PinTester(int[] user, int[] valid)
   {
      // Create an array to hold a copy of the
      // PIN digits entered by the user.
      userPIN = new int[user.length];

      // Copy the user array to the userPIN array.
      for (int i = 0; i < user.length; i++)
         userPIN[i] = user[i];

      // Create an array to hold a copy of the
      // valid PIN digits.
   validPIN = new int[valid.length];

   // Copy the valid array to the validPIN array.
   for (int i = 0; i < valid.length; i++)
      validPIN[i] = valid[i];
}

/**
    The isValid method compares the PIN digits entered
    by the user and determines whether they are valid.
    @return true if valid, false otherwise.
*/

public boolean isValid()
{
   int i;           // Loop control variable
   boolean isEqual; // Flag

   // Compare the lengths of the two arrays. If
   // the arrays are not the same size, then they
   // are not equal.

   if (userPIN.length != validPIN.length)
       isEqual = false;
   else // Compare the contents of the two arrays
   {
      i = 0;
      isEqual = true;

      while (isEqual && i < userPIN.length)
      {
         if (userPIN[i] != validPIN[i])
             isEqual = false;
         i++;
       }
     }

     return isEqual;
   }
}
```

Because you have only been asked to write a method that performs the comparison between the customer's input and the PIN that was retrieved from the database, you will also need to write a program that tests the method. Code Listing CS3-2 shows the complete program.

**Code Listing CS3-2** (PinTesterDemo.java)

```
/**
   This program tests the PinTester class with
   various PIN values.
*/

public class PinTesterDemo
{
   public static void main(String [] args)
   {
      // Create two sample arrays that are identical.
      int [] pin1 = {2, 4, 1, 8, 7, 9, 0};
      int [] pin2 = {2, 4, 1, 8, 7, 9, 0};

      // Create two sample arrays that are different.
      int [] pin3 = {2, 4, 6, 8, 7, 9, 0};
      int [] pin4 = {1, 2, 3, 4, 5, 6, 7};

      // Create two PinTester objects.
      PinTester pt1 = new PinTester(pin1, pin2);
      PinTester pt2 = new PinTester(pin3, pin4);

      // Compare pin1 and pin2. These arrays are identical.
      if (pt1.isValid())
      {
         System.out.println(“SUCCESS: pin1 and pin2 ” +
                            “are the same.”);
      }
      else
      {
         System.out.println(“ERROR: pin1 and pin2 ” +
                            “are different.”);
      }

      // Compare pin3 and pin4. All of the elements in these
      // arrays are different.

      if (pt2.isValid())
      {
          System.out.println(“ERROR: pin3 and pin4 ” +
                             “are the same.”);
      }
      else
      {
          System.out.println(“SUCCESS: pin3 and pin4 ” +
                             “are different.”);
```

```
        }
    }
}
```

**Program Output**

```
SUCCESS: pin1 and pin2 are the same.
SUCCESS: pin3 and pin4 are different.
```

CASE STUDY

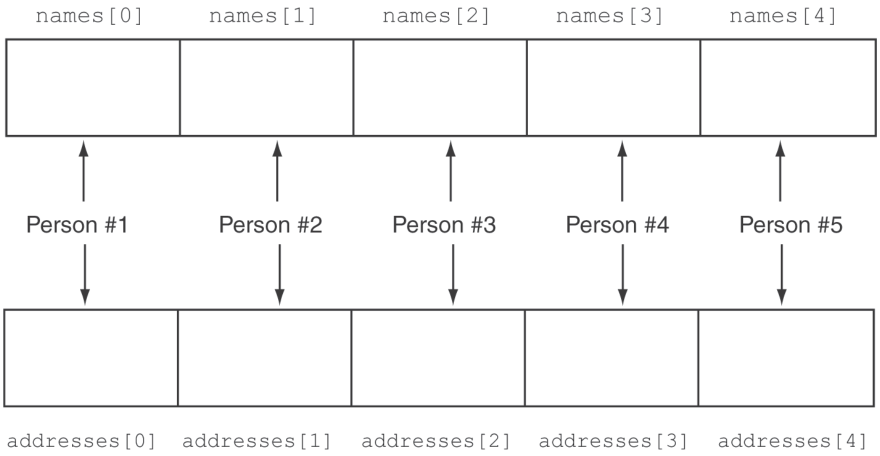
# 4 Parallel Arrays

Sometimes it's useful to store related data in two or more arrays. For example, assume a program uses the following arrays:

```
String[] names = new String[5];
String[] addresses = new String[5];
```

The `names` array is used to store the names of five persons, and the `addresses` array is used to store the addresses of the same five persons. The data for one person is stored in the same relative location in each array. For example, the first person's name is stored at `names[0]` and that same person's address is stored at `addresses[0]`. This relationship between the arrays is illustrated in Figure CS4-1.

**Figure CS4-1** The relationship between the names and addresses arrays


names[0]
names[1]
names[2]
names[3]
names[4]
Person #1
Person #2
Person #3
Person #4
Person #5
addresses[0]
addresses[1]
addresses[2]
addresses[3]
addresses[4]

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

To access the data, use the same subscript with both arrays. For example, the following loop displays each person's name and address.

```
for (int i = 0; i < names.length; i++)
{
   System.out.println("Name: " + names[i]);
   System.out.println("Address: " + addresses[i]);
}
```

The names and addresses arrays are examples of parallel arrays. *Parallel arrays* are two or more arrays that hold related data, and the related elements of each array are accessed with a common subscript.

Parallel arrays are especially useful when the related data is of unlike types. For example, the program in Code Listing Code Listing CS4-1 is a payroll program. It uses two arrays: one to store the hours worked by each employee (as `int` values), and another to store each employee's hourly pay rate (as `double` values).

**Code Listing CS4-1** (`ParallelArrays.java`)

```
import java.util.Scanner;

/**
   This program demonstrates parallel arrays.
*/

public class ParallelArrays
{
   public static void main(String [] args)
   {
       final int NUM_EMPLOYEES = 3;
       int[] hours = new int[NUM_EMPLOYEES];
       double[] payRates = new double[NUM_EMPLOYEES];

       // Get the hours worked by each employee.
       getPayData(hours, payRates);

       // Display each employee's gross pay.
       displayGrossPay(hours, payRates);
   }

 /**
    The getPayData method accepts as arguments arrays
    for employees' hours and pay rates. The user enters
    values for these arrays.
 */
```

```

   private static void getPayData(int[] hours, double[] payRates)
   {
      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get each employee’s hours worked and pay rate.
      for (int i = 0; i < hours.length; i++)
      {
         // Get the hours worked for an employee.
         System.out.printf(
            “Enter the hours worked by employee #%d: ”,
            (i + 1));
          hours[i] = keyboard.nextInt();

          // Get the hourly pay rate for this employee.
          System.out.printf(
             “Enter the hourly pay rate for employee #%d: ”,
             (i + 1));
           payRates[i] = keyboard.nextDouble();
     }
   }

   /**
      The displayGrossPay method accepts as arguments
      arrays for employees’ hours and pay rates. The
      method uses these arrays to calculate and display
      each employee’s gross pay.
   */

   private static void displayGrossPay(int [] hours,
                                       double [] payRates)
   {
      double grossPay; // To hold gross pay

      // Display each employee’s gross pay.
      for (int i = 0; i < hours.length; i++)
      {
         // Calculate the gross pay.
         grossPay = hours[i] * payRates[i];

         // Display the gross pay.
         System.out.printf(
              “The gross pay for employee #%d is $%,.2f.\n”,
              (i + 1), grossPay);
      }
    }
  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the hours worked by employee #1:  15 [Enter]
Enter the hourly pay rate for employee  #1:  12.00 [Enter]
Enter the hours worked by employee #2:  20 [Enter]
Enter the hourly pay rate for employee  #2:  18.00 [Enter]
Enter the hours worked by employee #3:  40 [Enter]
Enter the hourly pay rate for employee  #3:  25.50 [Enter]
The gross pay for employee #1 is $180.00.
The gross pay for employee #2 is $360.00.
The gross pay for employee #3 is $1,020.00.
```

CASE STUDY

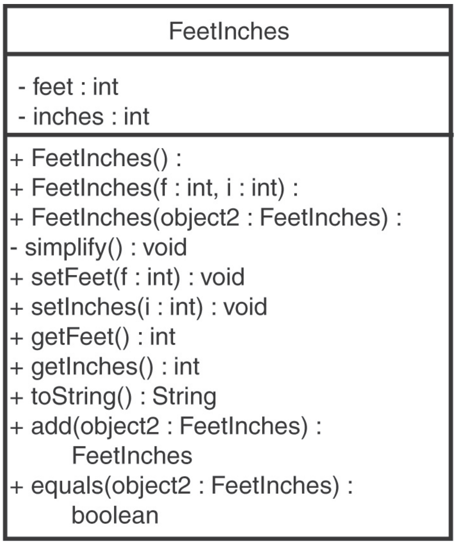
# 5 The FeetInches Class

Recall from Chapter 3 that the `String` class has a method, `equals`, that accepts a `String` object as an argument. As a review, here is an example of how the code can be used:

```
String name1 = “Joe”;
String name2 = “Jo”;
if (name1.equals(name2))
   System.out.println(“The names are the same.”);
else
   System.out.println(“The names are not the same.”);
```

This is an example of a *same class operation*, where an object performs an operation involving another object of the same class. In this section we use a class named `FeetInches` to study same class operations. The `FeetInches` class is designed to hold distances or measurements expressed in feet and inches. Figure CS5-1 shows a UML diagram for the class.

**Figure CS5-1** UML diagram for the `FeetInches` class


FeetInches
- feet : int
- inches : int
+ FeetInches() :
+ FeetInches(f : int, i : int) :
+ FeetInches(object2 : FeetInches) :
- simplify() : void
+ setFeet(f : int) : void
+ setInches(i : int) : void
+ getFeet() : int
+ getInches() : int
+ toString() : String
+ add(object2 : FeetInches) : FeetInches
+ equals(object2 : FeetInches) : boolean

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

The class has two `int` fields: `feet` and `inches`. Together these fields hold a distance measured in feet and inches, such as 12 feet 7 inches. The `feet` field holds the feet part and the `inches` field holds the inches part. Table CS5-1 describes the class's methods.

**Table CS5-1** The FeetInches class methods

| Method | Description |
|---|---|
| Constructor #1 (no-arg constructor) | This constructor assigns 0 to both the `feet` and `inches` fields. |
| Constructor #2 | This constructor accepts two `int` arguments that are assigned to the `feet` and `inches` fields. The `simplify` method is also called. |
| Constructor #3 | This constructor accepts a `FeetInches` object as its argument. The object that is being created will become a copy of the object passed as an argument. This type of constructor is sometimes referred to as a "copy constructor." |
| `simplify` | This method adjusts any set of values where the `inches` field is greater than 11. For example, 3 feet 14 inches would be adjusted to read 4 feet 2 inches. |
| `setFeet` | This method accepts an `int` argument that is assigned to the `feet` field. |
| `setInches` | This method accepts an `int` argument that is assigned to the `inches` field. The `simplify` method is then called. |
| `getFeet` | This method returns the value in the `feet` field. |
| `getInches` | This method returns the value in the `inches` field. |
| `toString` | This method returns a string representing the distance held by the object. For example, if an object's `feet` field holds 3 and its `inches` field holds 7, the `toString` method would return the string "3 feet 7 inches". |
| `add` | This method accepts a `FeetInches` object as its argument. It returns a reference to a `FeetInches` object that is the sum of the calling object and the object that was passed as the argument. |
| `equals` | This method accepts a `FeetInches` object as its argument. It returns the `boolean` value `true` if the calling object and the argument object hold the same data. Otherwise it returns `false`. |
| `copy` | This method returns a reference to a new `FeetInches` object that is a copy of the calling object. |

Before looking at the code for the `FeetInches` class, it might help to see a program that uses the class. The program in Code Listing CS5-1 demonstrates the class.

**Code Listing CS5-1** `(DistanceDemo.java)`

```java
import java.util.Scanner;

/**
   This program demonstrates the FeetInches class.
*/

public class DistanceDemo
{
   public static void main(String[] args)
   {
      int feet, inches;          // To hold feet and inches
      FeetInches distance;       // To reference a FeetInches object

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get a distance in feet and inches.
      System.out.println(“Enter a distance in feet ”
                          + “and inches.”);
      System.out.print(“First enter the feet: ”);
      feet = keyboard.nextInt();
      System.out.print(“Now enter the inches: ”);
      inches = keyboard.nextInt();

      // Create a FeetInches object, initialized with
      // the values entered by the user.
      distance = new FeetInches(feet, inches);

      // Display the object’s contents.
      System.out.println(“The distance you entered is ”
                          + distance.toString());
  }
}
```

**Program Output with Example Input Shown in Bold**

```
Enter a distance in feet and inches.
First enter the feet: 5 [Enter]
Now enter the inches: 7 [Enter]
The distance you entered is 5 feet 7 inches
```

**Program Output with Example Input Shown in Bold**

```
Enter a distance in feet and inches.
First enter the feet: 5 [Enter]
Now enter the inches: 14 [Enter]
The distance you entered is 6 feet 2 inches
```

Notice that the `FeetInches` class automatically adjusts measurements so that the `inches` field is never greater than 11. For example, the distance 5 feet 14 inches was adjusted to the more proper measurement 6 feet 2 inches. This adjustment is performed by a private method named `simplify`, which is called from the constructor and the `setInches` method.

Also notice that the class has a `toString` method which returns a string representing the distance held by an object. This method is useful for displaying a distance. Code Listing CS5-2 shows the `FeetInches` class.

**Code Listing CS5-2** (`FeetInches.java`)

```
/**
   The FeetInches class holds distances measured in
   feet and inches.
*/

public class FeetInches
{
   private int feet;    // The number of feet
   private int inches;  // The number of inches

   /**
      This constructor assigns 0 to the feet
      and inches fields.
   */

   public FeetInches()
   {
      feet = 0;
      inches = 0;
   }

   /**
      This constructor accepts two arguments which
      are assigned to the feet and inches fields.
      The simplify method is then called.
      @param f The value to assign to feet.
      @param i The value to assign to inches.
   */

   public FeetInches(int f, int i)
   {
      feet = f;
      inches = i;
      simplify();
   }

   /**
      The following is a copy constructor. It accepts a
      reference to another FeetInches object. The feet
      and inches fields are set to the same values as
      those in the argument object.
      @param object2 The object to copy.
   */

   public FeetInches(FeetInches object2)
   {
      feet = object2.feet;
      inches = object2.inches;
   }

   /**
      The simplify method adjusts the values
      in feet and inches to conform to a
      standard measurement.
   */

   private void simplify()
   {
      if (inches > 11)
      {
         feet = feet + (inches / 12);
         inches = inches % 12;
      }
   }

   /**
      The setFeet method assigns a value to
      the feet field.
      @param f The value to assign to feet.
```

```
*/

public void setFeet(int f)
{
   feet = f;
}

/**
   The setInches method assigns a value to
   the inches field.
   @param i The value to assign to inches.
*/

public void setInches(int i)
{
   inches = i;
   simplify();
}

/**
   getFeet method
   @return The value in the feet field.
*/

public int getFeet()
{
   return feet;
}

/**
   getInches method
   @return The value in the inches field.
*/

public int getInches()
{
   return inches;
}

/**
   toString method
   @return a reference to a String stating
   the feet and inches.
 */

public String toString()
```

```
   {
      String str = feet + “ feet ” +
                   inches + “ inches”;
      return str;
   }

   /**
      The add method returns a FeetInches object
      that holds the sum of this object and another
      FeetInches object.
      @param object2 The other FeetInches object.
      @return A reference to a FeetInches object.
   */

   public FeetInches add(FeetInches object2)
   {
      int totalFeet,       // To hold the sum of feet
          totalInches;     // To hold the sum of inches

      totalFeet = feet + object2.feet;
      totalInches = inches + object2.inches;
      return new FeetInches(totalFeet, totalInches);
   }

   /**
      The equals method compares this object to the
      argument object. If both have the same values,
      the method returns true.
      @return true if the objects are equal, false
      otherwise.
   */

   public boolean equals(FeetInches object2)
   {
      boolean status;

      if (object2 == null)
         status = false;
      else if (feet == object2.feet &&
               inches == object2.inches)
         status = true;
      else
         status = false;
      return status;
   }

   /**
      The copy method makes a copy of the
      calling object.
      @return A reference to the copy.
   */

   public FeetInches copy()
   {
      // Make a new FeetInches object and
      // initialize it with the same data
      // as the calling object.
      FeetInches newObject =
               new FeetInches(feet, inches);

      // Return a reference to the new object.
      return newObject;
   }
}
```

## More about the `toString` Method

The program in Code Listing CS5-1 explicitly called the `FeetInches` class's `toString` method to display the contents of the `distance` object. The following statement appears in lines 30 and 31:

```
System.out.println(“The distance you entered is ”
                   + distance.toString());
```

In actuality, it is unnecessary to explicitly call the `toString` method in this statement. If you write a `toString` method for a class, Java will automatically call the method any time you concatenate an object of the class with a string. For example, the following statement would implicitly call the `distance` object's `toString` method:

```
System.out.println(“The distance you entered is ”
                   + distance);
```

Java also implicitly calls an object's `toString` method when the object is passed to the `print` or `println` method. Assuming that `distance` is a `FeetInches` object, here is an example:

```
System.out.println(distance);
```

This statement is equivalent to:

```
System.out.println(distance.toString());
```

**NOTE:** Every class automatically has a `toString` method that returns a string containing the object's class name, followed by the `@` symbol, followed by an integer that is usually based on the object's memory address. This method is called when necessary if you have not provided your own `toString` method. This is discussed further in Chapter 8.

## The add Method

The `add` method is the first `FeetInches` method we will study that performs a same class operation. It allows us to add one `FeetInches` object to another. Let's take a closer look at the method, which appears in lines 130 through 138. For your convenience, the method is shown here:

```
130   public FeetInches add(FeetInches object2)
131   {
132      int totalFeet,    // To hold the sum of feet
133          totalInches;  // To hold the sum of inches
134
135      totalFeet = feet + object2.feet;
136      totalInches = inches + object2.inches;
137      return new FeetInches(totalFeet, totalInches);
138   }
```

This method accepts a `FeetInches` object as its argument. The parameter variable `object2` will reference the object that was passed as an argument. The statement in line 135 adds the `feet` field of the calling object to the `feet` field of the argument object, and stores the result in the local variable `totalFeet`. Notice that the method can access `object2`'s `feet` field directly. Because the `add` method is a member of the `FeetInches` class, it is allowed to access `object2`'s private fields.

Next, in line 136, the method adds the `inches` field of the calling object to the `inches` field of `object2`, and stores the result in the local variable `totalInches`.

The last statement, in line 137, creates a new `FeetInches` object, passes `totalFeet` and `totalInches` to the constructor, and returns a reference to the object. The program in Code Listing CS5-3 demonstrates the method.

**Code Listing CS5-3** (`DistanceAdd.java`)

```
/**
   This program uses the FeetInches class's add method
   to add two distances.
*/

public class DistanceAdd
{
   public static void main(String[] args)
   {
      // Create a FeetInches object holding 5 feet, 9 inches.
      FeetInches distance1 = new FeetInches(5, 9);

      // Create a FeetInches object holding 2 feet, 5 inches.
      FeetInches distance2 = new FeetInches(2, 5);

      // Declare a FeetInches variable that does not
      // reference an object.
```

```
        FeetInches distance3;

        // Add distance 1 and distance 2, store the
        // result in distance3.
        distance3 = distance1.add(distance2);

        // Display the results.
        System.out.println(“The first distance is ” + distance1);
        System.out.println(“The second distance is ” + distance2);
        System.out.println(“The sum of these distances is ” + distance3);
    }
}
```

**Program Output**

```
The first distance is 5 feet 9 inches
The second distance is 2 feet 5 inches
The sum of these distances is 8 feet 2 inches
```

## The `equals` Method

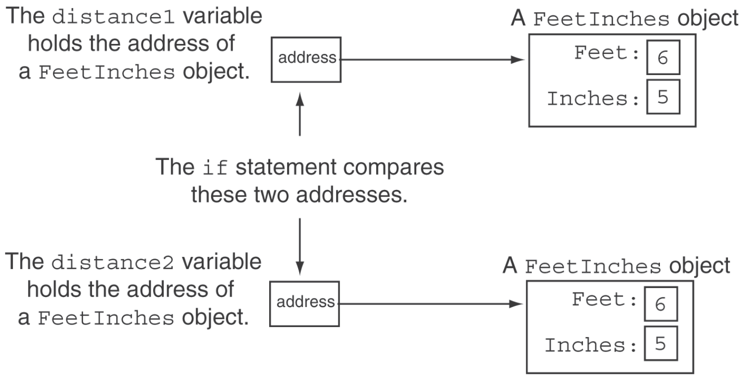
Now let’s take a closer look at the `equals` method, which also performs a same class operation: It compares the contents of two `FeetInches` objects and determines whether the two objects are equal. This method is useful because you cannot use the == operator to compare the contents of two objects. For example, the following code appears to compare two `FeetInches` objects, but in reality does not.

```
FeetInches distance1 = new FeetInches(6, 5);
FeetInches distance2 = new FeetInches(6, 5);
if (distance1 == distance2)        // This is a mistake.
   System.out.println(“The objects are the same.”);
else
   System.out.println(“The objects are not the same.”);
```

When you use the == operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables. This is illustrated in Figure CS5-2.

Because the two variables reference different objects in memory, they will contain different addresses. Therefore, the result of the `boolean` expression `distance1 == distance2` is `false` and the code reports that the objects are not the same. Instead of using the == operator to compare two `FeetInches` objects, we should use the `equals` method.

**Figure CS5-2** The `if` statement tests the contents of the reference variables, not the contents of the objects the variables reference


The distance1 variable holds the address of a FeetInches object.
address
A FeetInches object
Feet: 6
Inches: 5
The if statement compares these two addresses.
The distance2 variable holds the address of a FeetInches object.
address
A FeetInches object
Feet: 6
Inches: 5

Here is the code for the method, which appears in lines 148 through 160:

```
148    public boolean equals(FeetInches object2)
149    {
150       boolean status;
151
152       if (object2 == null)
153          status = false;
154        else if (feet == object2.feet &&
155                 inches == object2.inches)
156            status = true;
157         else
158            status = false;
159         return status;
160     }
```

This method accepts a `FeetInches` object as its argument. The parameter variable `object2` will reference the object that was passed as an argument. Lines 154 and 155 perform the following comparison: If the `feet` field of the calling object is equal to the `feet` field of `object2`, and the `inches` field of the calling object is equal to the `inches` field of `object2`, then the two objects contain the same distance. In this case, the local variable `status` (a `boolean`) is set to `true`. Otherwise, `status` is set to `false`. The method returns the value of the `status` variable in line 159. The program in Code Listing CS5-4 demonstrates the `equals` method.

**Code Listing CS5-4** (`DistanceCompare.java`)

```
/**
  This program uses the FeetInches class's equals
  method to compare two distances.
*/
```

```

public class DistanceCompare
{
   public static void main(String[] args)
   {
      // Create a FeetInches object holding 5 feet, 9 inches.
      FeetInches distance1 = new FeetInches(5, 9);

      // Create a FeetInches object holding 5 feet, 9 inches.
      FeetInches distance2 = new FeetInches(5, 9);

     // Create a FeetInches object holding 7 feet, 4 inches.
     FeetInches distance3 = new FeetInches(7, 4);

     // Display the distances.
     System.out.println(“The first distance is ” + distance1);
     System.out.println(“The second distance is ” + distance2);
     System.out.println(“The third distance is ” + distance3);

     // Compare distance1 and distance2.
     if (distance1.equals(distance2))
        System.out.println(“distances 1 and 2 are the same.”);
     else
         System.out.println(“distances 1 and 2 are not the same.”);

     // Compare distance1 and distance3.
     if (distance1.equals(distance3))
        System.out.println(“distances 1 and 3 are the same.”);
     else
         System.out.println(“distances 1 and 3 are not the same.”);
  }
}
```

**Program Output**

```
The first distance is 5 feet 9 inches
The second distance is 5 feet 9 inches
The third distance is 7 feet 4 inches
distances 1 and 2 are the same.
distances 1 and 3 are not the same.
```

If you want to be able to compare the objects of a given class, you should always write an `equals` method for the class.

**NOTE:** Every class automatically has an `equals` method, which works the same as the == operator. This method is called when necessary if you have not provided your own `equals` method.This is discussed further in Chapter 10.

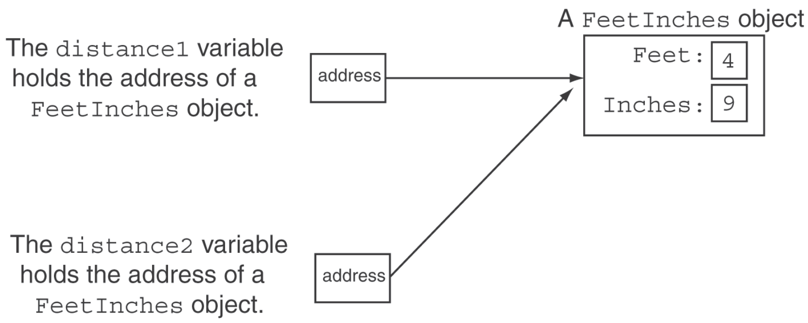
## The `copy` Method

The `copy` method also performs a same class operation: It returns a `FeetInches` object that is a copy of the calling object. This method is necessary because you cannot copy objects with a simple assignment statement, as you would with primitive variables. For example, look at the following code:

```
FeetInches distance1 = new FeetInches(4, 9);
FeetInches distance2 = distance1;
```

The first statement creates a `FeetInches` object and assigns its address to the `distance1` variable. The second statement assigns `distance1` to `distance2`. This does not make a copy of the object referenced by `distance1`. Rather, it makes a copy of the address that is stored in `distance1` and stores it in `distance2`. After this statement executes, both the `distance1` and `distance2` variables will reference the same object. This is illustrated in Figure CS5-3.

**Figure CS5-3** Both variables reference the same object


A FeetInches object
The distance1 variable holds the address of a FeetInches object.
address
Feet: 4
Inches: 9
The distance2 variable holds the address of a FeetInches object.
address

The copy method, which appears in lines 168 through 178, creates a new `FeetInches` object and passes the calling object's `feet` and `inches` fields as arguments to the constructor. This makes the new object a copy of the calling object. The method then returns a reference to the new object. Here is the method's code:

```
168    public FeetInches copy()
169    {
170       // Make a new FeetInches object and
171       // initialize it with the same data
172       // as the calling object.
173       FeetInches newObject =
174                new FeetInches(feet, inches);
175
176        // Return a reference to the new object.
177        return newObject;
178    }
```

## The Copy Constructor

The `FeetInches` class has three constructors: a no-arg constructor, a constructor that accepts arguments for the `feet` and `inches` fields, and a third constructor that accepts a `FeetInches` object as its argument. Here is the code for the third constructor, which appears in lines 45 through 49:

```
45   public FeetInches(FeetInches object2)
46   {
47      feet = object2.feet;
48      inches = object2.inches;
49   }
```

This type of constructor is sometimes referred to as a *copy constructor*. It makes the object that is being created a copy of the object that was passed as an argument. For example, the following code segment creates two `FeetInches` objects. The second one is a copy of the first one.

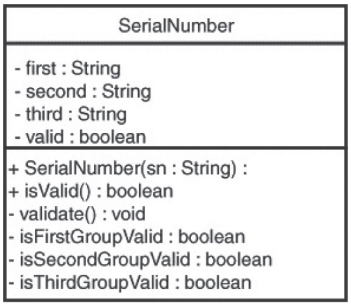
```
// Create a FeetInches object holding 5 feet, 9 inches.
FeetInches distance1 = new FeetInches(5, 9);
// Create another one that is a copy.
FeetInches distance2 = new FeetInches(distance1);
```

# CASE STUDY 6

# The `SerialNumber` Class

In this case study you examine the `SerialNumber` class, which is used by the Home Software Company to validate software serial numbers. A valid software serial number is in the form `LLLLL-DDDD-LLLL`, where `L` indicates an alphabetic letter and `D` indicates a numeric digit. For example, `WRXTQ-7786-PGVZ` is a valid serial number. Notice that a serial number consists of three groups of characters, delimited by hyphens. Figure CS6-1 shows a UML diagram for the `SerialNumber` class.

**Figure CS6-1** UML diagram for the `SerialNumber` class


SerialNumber
- first : String
- second : String
- third : String
- valid : boolean
+ SerialNumber(sn : String) :
+ isValid() : boolean
- validate() : void
- isFirstGroupValid : boolean
- isSecondGroupValid : boolean
- isThirdGroupValid : boolean

The fields `first`, `second`, and `third` are used to hold the first, second, and third groups of characters in a serial number. The `valid` field is set to `true` by the constructor to indicate a valid serial number, or `false` to indicate an invalid serial number. Table CS6-1 describes the class's methods.

Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

**Table CS6-1** `SerialNumber` class's methods

| Method | Description |
|---|---|
| Constructor | The constructor accepts a string argument that contains a serial number. The string is tokenized and its tokens are stored in the `first`, `second`, and `third` fields. The `validate` method is called. |
| `isValid` | This method returns the value in the `valid` field. |
| `validate` | This method calls the `isFirstGroupValid`, `isSecondGroupValid`, and `isThirdGroupValid` methods to validate the `first`, `second`, and `third` fields. |
| `isFirstGroupValid` | This method returns `true` if the value stored in the `first` field is valid. Otherwise, it returns `false`. |
| `isSecondGroupValid` | This method returns `true` if the value stored in the `second` field is valid. Otherwise, it returns `false`. |
| `isThirdGroupValid` | This method returns `true` if the value stored in the `third` field is valid. Otherwise, it returns `false`. |

The code for the `SerialNumber` class is shown in Code Listing CS6-1.

**Code Listing CS6-1** (`SerialNumber.java`)

```java
/**
   The SerialNumber class takes a software serial number in
   the form of LLLLL-DDDD-LLLL where each L is a letter
   and each D is a digit. The serial number has three groups
   of characters, separated by hyphens. The class extracts
   the three groups of characters and validates them.
*/

public class SerialNumber
{
   private String first;  // First group of characters
   private String second; // Second group of characters
   private String third;  // Third group of characters
   private boolean valid; // Flag indicating validity

   /**
      The constructor breaks a serial number into
      three groups and each group is validated.
      @param sn A serial number.
   */

   public SerialNumber(String sn)
   {
      // Trim the argument.
      sn = sn.trim();

      // Tokenize the argument.
      String tokens[] = sn.split("-");

      // Validate the tokens.
      if (tokens.length != 3)
         valid = false;
      else
      {
         first = tokens[0];
         second = tokens[1];
         third = tokens[2];
         validate();
     }

   }

   /**
      The isValid method returns a boolean value
      indicating whether the serial number is valid.
      @return true if the serial number is valid,
              otherwise false.
   */

   public boolean isValid()
   {
      return valid;
   }

   /**
      The validate method sets the valid field to true
      if the serial number is valid. Otherwise it sets
      valid to false.
   */

   private void validate()
   {
      if (isFirstGroupValid() && isSecondGroupValid() &&
          isThirdGroupValid())
         valid = true;
      else
         valid = false;
   }

   /**
      The isFirstGroupValid method validates the first
   group of characters.
   @return true if the group is valid, otherwise false.
*/

private boolean isFirstGroupValid()
{
   boolean groupGood = true; // Flag
   int i = 0;         // Loop control variable

   // Check the length of the group.
   if (first.length() != 5)
      groupGood = false;

   // See if each character is a letter.
   while (groupGood && i < first.length())
   {
      if (!Character.isLetter(first.charAt(i)))
         groupGood = false;
      i++;
   }

   return groupGood;
}

/**
   The isSecondGroupValid method validates the second
   group of characters.
   @return true if the group is valid, otherwise false.
*/


private boolean isSecondGroupValid()
{
   boolean groupGood = true; // Flag
   int i = 0;                // Loop control variable

   // Check the length of the group.
   if (second.length() != 4)
      groupGood = false;

   // See if each character is a digit.
   while (groupGood && i < second.length())
   {
      if (!Character.isDigit(second.charAt(i)))
         groupGood = false;
      i++;
   }

120       return groupGood;
121    }
122 
123    /**
124       The isThirdGroupValid method validates the third
125       group of characters.
126       @return true if the group is valid, otherwise false.
127    */
128 
129 
130    private boolean isThirdGroupValid()
131    {
132       boolean groupGood = true;  // Flag
133       int i = 0;                 // Loop control variable
134 
135       // Check the length of the group.
136       if (third.length() != 4)
137          groupGood = false;
138 
139       // See if each character is a letter.
140       while (groupGood && i < third.length())
141       {
142          if (!Character.isLetter(third.charAt(i)))
143             groupGood = false;
144          i++;
145       }
146 
147       return groupGood;
148    }
149 }
```

Let's take a closer look at the constructor.

Notice that in line 25 we call the argument's `trim` method to remove any leading and/or trailing whitespace characters. This is important because whitespace characters are not used as delimiters in this code. If the argument contains leading whitespace characters, they will be included as part of the first token. Trailing whitespace characters will be included as part of the last token. Then, in line 27 we call the argument's `split` method to tokenize the string, using the "-" character as a delimiter. The tokens are returned to an array named `tokens`.

Next, the `if` statement in lines 31 through 39 executes:

```
31     if (tokens.length != 3)
32        valid = false;
33     else
34     {
35        first = tokens[0];
36        second = tokens[1];
37        third = tokens[2];
38        validate();
39     }
```

A valid serial number must have three groups of characters, so the `if` statement determines whether the `tokens` array has three elements. If not, the `valid` field is set to `false`. Otherwise, the three tokens are assigned to the `first`, `second`, and `third` fields. Last, the `validate` method is called. The `validate` method calls the `isFirstGroupValid`, `isSecondGroupValid`, and `isThirdGroupValid` methods to validate the three groups of characters. In the end, the `valid` field will be set to `true` if the serial number is valid, or `false` otherwise. The program in Code Listing CS6-2 demonstrates the class.

**Code Listing CS6-2** (`SerialNumberTester.java`)

```
/**
   This program demonstrates the SerialNumber class.
*/

public class SerialNumberTester
{
   public static void main(String[] args)
   {
      String serial1 = “GHTRJ-8975-AQWR”; // Valid
      String serial2 = “GHT7J-8975-AQWR”; // Invalid
      String serial3 = “GHTRJ-8J75-AQWR”; // Invalid
      String serial4 = “GHTRJ-8975-AQ2R”; // Invalid

      // Validate serial1.

      SerialNumber sn = new SerialNumber(serial1);
      if (sn.isValid())
         System.out.println(serial1 + “ is valid.”);
      else
         System.out.println(serial1 + “ is invalid.”);

      // Validate serial2.

      sn = new SerialNumber(serial2);
      if (sn.isValid())
         System.out.println(serial2 + “ is valid.”);
         else
            System.out.println(serial2 + “ is invalid.”);

         // Validate serial3.

         sn = new SerialNumber(serial3);
         if (sn.isValid())
            System.out.println(serial3 + “ is valid.”);
         else
            System.out.println(serial3 + “ is invalid.”);

         // Validate serial4.

         sn = new SerialNumber(serial4);
         if (sn.isValid())
            System.out.println(serial4 + “ is valid.”);
         else
            System.out.println(serial4 + “ is invalid.”);
     }
   }
```

**Program Output**

```
GHTRJ-8975-AQWR is valid.
GHT7J-8975-AQWR is invalid.
GHTRJ-8J75-AQWR is invalid.
GHTRJ-8975-AQ2R is invalid.
```

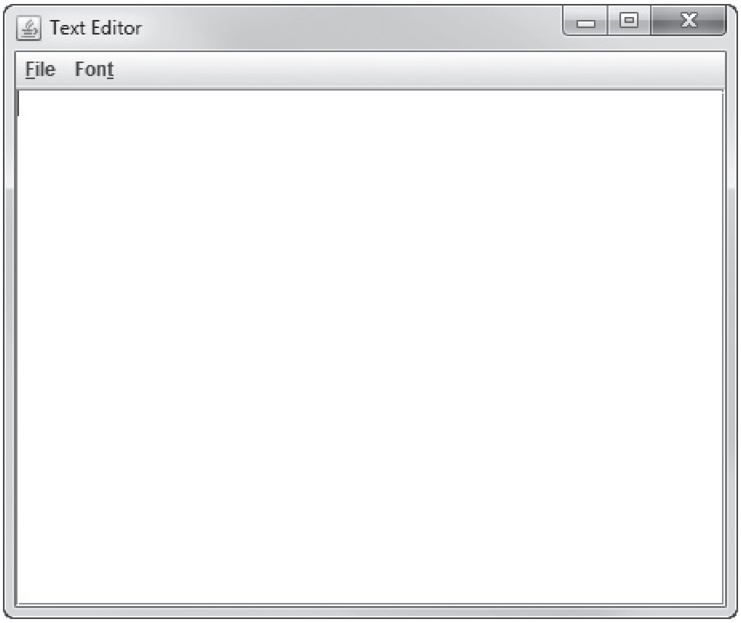
CASE STUDY

# 7

# A Simple Text Editor Application

To demonstrate the `JTextArea` component, fonts, menus, and file choosers, we present a simple text editor application. This application allows you to create new text files and open existing text files. The file contents are displayed in a text area. You can also change the font and style of the text that is displayed in the text area. The application's window is shown in Figure CS7-1. Line wrapping is turned on, using word wrap style, and the text area is in a scroll pane.

**Figure CS7-1** The text editor window

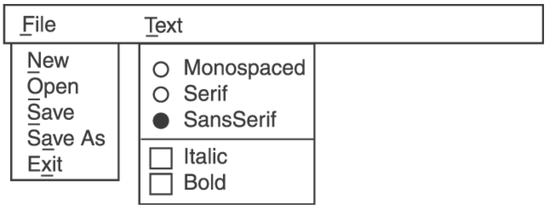


Access online material for this text at www.pearson.com/gaddis

From *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. Tony Gaddis.
Copyright © 2016 by Pearson Education, Inc. All rights reserved.

The application uses a menu system to perform these operations, which is shown in Figure CS7-2.

**Figure CS7-2** Menu system for the text editor application



Each menu item generates an action event that is handled by an action listener. The following section presents a summary of the actions performed by each menu item.

## File Menu

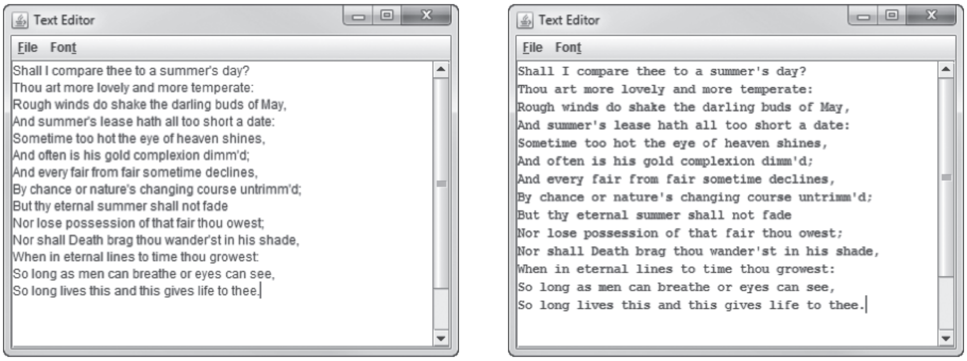
| | |
|---|---|
| New | This menu item clears any text that is stored in the text area. In addition, the class's `filename` field is set to `null`. The `filename` field contains the path and name of the file that is currently displayed in the text area. |
| Open | This menu item displays a file chooser that allows the user to select a file to open. If the user selects a file, it is opened and its contents are read into the text area. The path and name of the file are stored in the `filename` field. |
| Save | This menu item saves the contents of the text area. The contents are saved to the file with the name and path stored in the `filename` field. If the `filename` field is set to `null`, which would indicate that the file has not been saved yet, then this menu item performs the same action as the Save As menu item. |
| Save As | This menu item displays a file chooser that allows the user to select a location and file name. The contents of the text area are written to the selected file, and the path and file name are stored in the `filename` field. (Be careful when using this menu item. As it is currently written, this application does not warn you when you are about to overwrite an existing file!) |
| Exit | This menu item ends the application. |

## Font Menu

| | |
|---|---|
| Monospaced | This radio button menu item changes the text area's font to Monospaced. |
| Serif | This radio button menu item changes the text area's font to Serif. |
| SansSerif | This radio button menu item changes the text area's font to SansSerif. |
| Italic | This check box menu item changes the text area's style to italic. |
| Bold | This check box menu item changes the text area's style to bold. |

Code Listing CS7-1 shows the code for the `TextEditor` class. The `main` method creates an instance of the class, which displays the text editor window. Figure CS7-3 shows the window displaying text in various fonts and styles.

**Figure CS7-3** Text displayed in various fonts and styles



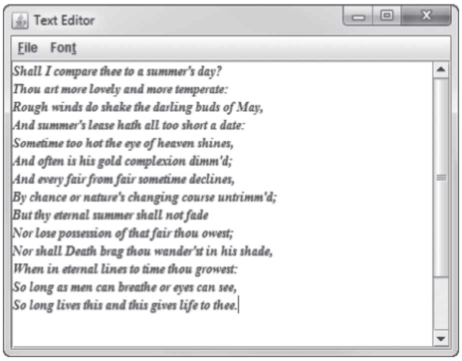
SansSerif font, plain style

Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.

Monospaced font, bold style

Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.

Serif font, bold and italic style



***Shall I compare thee to a summer's day?***
***Thou art more lovely and more temperate:***
***Rough winds do shake the darling buds of May,***
***And summer's lease hath all too short a date:***
***Sometime too hot the eye of heaven shines,***
***And often is his gold complexion dimm'd;***
***And every fair from fair sometime declines,***
***By chance or nature's changing course untrimm'd;***
***But thy eternal summer shall not fade***
***Nor lose possession of that fair thou owest;***
***Nor shall Death brag thou wander'st in his shade,***
***When in eternal lines to time thou growest:***
***So long as men can breathe or eyes can see,***
***So long lives this and this gives life to thee.***

**Code Listing CS7-1** `(TextEditor.java)`

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.util.Scanner;

/**
   The TextEditor class is a simple text editor.
*/

public class TextEditor extends JFrame
{
   // The following are fields for the menu system.
   // First, the menu bar
   private JMenuBar menuBar;

   // The menus
   private JMenu fileMenu;
   private JMenu fontMenu;

   // The menu items
   private JMenuItem newItem;
   private JMenuItem openItem;
   private JMenuItem saveItem;
   private JMenuItem saveAsItem;
   private JMenuItem exitItem;

   // The radio button menu items
   private JRadioButtonMenuItem monoItem;
   private JRadioButtonMenuItem serifItem;
   private JRadioButtonMenuItem sansSerifItem;

   // The checkbox menu items
   private JCheckBoxMenuItem italicItem;
   private JCheckBoxMenuItem boldItem;

   private String filename;   // To hold the file name
   private JTextArea editorText;// To display the text
   private final int NUM_LINES = 20; // Lines to display
   private final int NUM_CHARS = 40; // Chars per line

   /**
      Constructor
   */

   public TextEditor()
   {
      // Set the title.
      setTitle(“Text Editor”);

      // Specify what happens when the close
      // button is clicked.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the text area.
      editorText = new JTextArea(NUM_LINES, NUM_CHARS);

      // Turn line wrapping on.
      editorText.setLineWrap(true);
      editorText.setWrapStyleWord(true);

      // Create a scroll pane and add the text area to it.
      JScrollPane scrollPane = new JScrollPane(editorText);

      // Add the scroll pane to the content pane.
      add(scrollPane);

      // Build the menu bar.
      buildMenuBar();

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMenuBar method creates a menu bar and
      calls the createFileMenu method to create the
      file menu.
   */

   private void buildMenuBar()
   {
      // Build the file and font menus.
      buildFileMenu();
      buildFontMenu();

      // Create the menu bar.
      menuBar = new JMenuBar();

      // Add the file and font menus to the menu bar.
      menuBar.add(fileMenu);
      menuBar.add(fontMenu);

      // Set the menu bar for this frame.
      setJMenuBar(menuBar);
   }

   /**
      The buildFileMenu method creates the file menu
      and populates it with its menu items.
   */

   private void buildFileMenu()
   {
      // Create the New menu item.
      newItem = new JMenuItem("New");
      newItem.setMnemonic(KeyEvent.VK_N);
      newItem.addActionListener(new NewListener());

      // Create the Open menu item.
      openItem = new JMenuItem("Open");
      openItem.setMnemonic(KeyEvent.VK_O);
      openItem.addActionListener(new OpenListener());

      // Create the Save menu item.
      saveItem = new JMenuItem(“Save”);
      saveItem.setMnemonic(KeyEvent.VK_S);
      saveItem.addActionListener(new SaveListener());

      // Create the Save As menu item.
      saveAsItem = new JMenuItem(“Save As”);
      saveAsItem.setMnemonic(KeyEvent.VK_A);
      saveAsItem.addActionListener(new SaveListener());

      // Create the Exit menu item.
      exitItem = new JMenuItem(“Exit”);
      exitItem.setMnemonic(KeyEvent.VK_X);
      exitItem.addActionListener(new ExitListener());

      // Create a menu for the items we just created.
      fileMenu = new JMenu(“File”);
      fileMenu.setMnemonic(KeyEvent.VK_F);

      // Add the items and some separator bars to the menu.
      fileMenu.add(newItem);
      fileMenu.add(openItem);
      fileMenu.addSeparator();// Separator bar
      fileMenu.add(saveItem);
      fileMenu.add(saveAsItem);
      fileMenu.addSeparator();// Separator bar
      fileMenu.add(exitItem);
   }

   /**
      The buildFontMenu method creates the font menu
         and populates it with its menu items.
   */

   private void buildFontMenu()
   {
      // Create the Monospaced menu item.
      monoItem = new JRadioButtonMenuItem(“Monospaced”);
      monoItem.addActionListener(new FontListener());

      // Create the Serif menu item.
      serifItem = new JRadioButtonMenuItem(“Serif”);
      serifItem.addActionListener(new FontListener());

   // Create the SansSerif menu item.
   sansSerifItem =
           new JRadioButtonMenuItem(“SansSerif”, true);
   sansSerifItem.addActionListener(new FontListener());

   // Group the radio button menu items.
   ButtonGroup group = new ButtonGroup();
   group.add(monoItem);
   group.add(serifItem);
   group.add(sansSerifItem);

   // Create the Italic menu item.
   italicItem = new JCheckBoxMenuItem(“Italic”);
   italicItem.addActionListener(new FontListener());

   // Create the Bold menu item.
   boldItem = new JCheckBoxMenuItem(“Bold”);
   boldItem.addActionListener(new FontListener());

   // Create a menu for the items we just created.
   fontMenu = new JMenu(“Font”);
   fontMenu.setMnemonic(KeyEvent.VK_T);

   // Add the items and some separator bars to the menu.
   fontMenu.add(monoItem);
   fontMenu.add(serifItem);
   fontMenu.add(sansSerifItem);
   fontMenu.addSeparator();// Separator bar
   fontMenu.add(italicItem);
   fontMenu.add(boldItem);
}

/**
   Private inner class that handles the event that
   is generated when the user selects New from
   the file menu.
*/

 private class NewListener implements ActionListener
 {
   public void actionPerformed(ActionEvent e)
  {
      editorText.setText(“”);
      filename = null;
  }
}

```

```
   /**
      Private inner class that handles the event that
      is generated when the user selects Open from
      the file menu.
   */

   private class OpenListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         int chooserStatus;

         JFileChooser chooser = new JFileChooser();
         chooserStatus = chooser.showOpenDialog(null);
         if (chooserStatus == JFileChooser.APPROVE_OPTION)
         {
            // Get a reference to the selected file.
            File selectedFile = chooser.getSelectedFile();

            // Get the path of the selected file.
            filename = selectedFile.getPath();

            // Open the file.
            if (!openFile(filename))
            {
               JOptionPane.showMessageDialog(null,
                                 “Error reading ” +
                                 filename, “Error”,
                                 JOptionPane.ERROR_MESSAGE);
            }
         }
      }

      /**
         The openFile method opens the file specified by
         filename and reads its contents into the text
         area. The method returns true if the file was
         opened and read successfully, or false if an
         error occurred.
         @param filename The name of the file to open.
      */

      private boolean openFile(String filename)
      {
         boolean success;
         String inputLine, editorString = “”;

         try
         {
            // Open the file.
            File file = new File(filename);
            Scanner inputFile = new Scanner(file);

            // Read the file contents into the editor.
            while (inputFile.hasNext())
            {
               // Read a line from the file.
               inputLine = inputFile.nextLine();

               // Append it to the string to display
               // in the editor.
               editorString = editorString +
                              inputLine + “\n”;
            }

            // Display the string that was read from the
            // file in the editor.
            editorText.setText(editorString);

            // Close the file.
            inputFile.close();

            // Indicate that everything went OK.
            success = true;
         }
         catch (IOException e)
         {
            // Something went wrong.
            success = false;
         }

         // Return our status.
         return success;
      }
   }

   /**
      Private inner class that handles the event that
      is generated when the user selects Save or Save
      As from the file menu.
   */

   private class SaveListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         int chooserStatus;

         // If the user selected Save As, or the contents
         // of the editor have not been saved, use a file
         // chooser to get the file name. Otherwise, save
         // the file under the current file name.

         if (e.getActionCommand() == "Save As"  ||
             filename == null)
         {
            JFileChooser chooser = new JFileChooser();
            chooserStatus = chooser.showSaveDialog(null);
            if (chooserStatus == JFileChooser.APPROVE_OPTION)
            {
               // Get a reference to the selected file.
               File selectedFile =
                              chooser.getSelectedFile();

               // Get the path of the selected file.
               filename = selectedFile.getPath();
            }
         }

         // Save the file.
         if (!saveFile(filename))
         {
            JOptionPane.showMessageDialog(null,
                                "Error saving " +
                                filename,
                                "Error",
                                JOptionPane.ERROR_MESSAGE);
         }
      }

      /**
         The saveFile method saves the contents of the
         text area to a file. The method returns true if
         the file was saved successfully, or false if an
         error occurred.
         @param filename The name of the file.
         @return true if successful, false otherwise.
      */

      private boolean saveFile(String filename)
      {
         boolean success;
         String editorString;
         PrintWriter outputFile;
      try
      {
          // Open the file.
          outputFile = new PrintWriter(filename);

          // Write the contents of the text area
          // to the file.
          editorString = editorText.getText();
          outputFile.print(editorString);

          // Close the file.
          outputFile.close();

          // Indicate that everything went OK.
          success = true;
      }
      catch (IOException e)
      {
          // Something went wrong.
          success = false;
      }

      // Return our status.
      return success;
   }
}

/**
   Private inner class that handles the event that
   is generated when the user selects Exit from
   the file menu.
*/

private class ExitListener implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      System.exit(0);
   }
}

/**
   Private inner class that handles the event that
   is generated when the user selects an item from
   the font menu.
*/
```

```
    private class FontListener implements ActionListener
    {
       public void actionPerformed(ActionEvent e)
       {
          // Get the current font.
          Font textFont = editorText.getFont();

          // Retrieve the font name and size.
          String fontName = textFont.getName();
          int fontSize = textFont.getSize();

          // Start with plain style.
          int fontStyle = Font.PLAIN;

          // Determine which font is selected.
          if (monoItem.isSelected())
             fontName = “Monospaced”;
          else if (serifItem.isSelected())
             fontName = “Serif”;
          else if (sansSerifItem.isSelected())
             fontName = “SansSerif”;

          // Determine whether italic is selected.
          if (italicItem.isSelected())
             fontStyle += Font.ITALIC;

          // Determine whether bold is selected.
          if (boldItem.isSelected())
             fontStyle += Font.BOLD;

          // Set the font as selected.
          editorText.setFont(new Font(fontName,
                                 fontStyle, fontSize));
       }
    }

    /**
       main method
    */

    public static void main(String[] args)
    {
          TextEditor te = new TextEditor();
    }
}
```